Contributions à la sécurité des Java Card

Julien Iguchi-Cartigny



Contributions à la sécurité des Java Card

Mémoire pour l'obtention de

l'Habilitation à diriger des recherches
Discipline : Informatique

par

JULIEN IGUCHI-CARTIGNY

présenté le
4 décembre 2014

Composition du jury :

Président :	GILLES GRIMAUD	Professeur - Université Lille 1 - LIFL
Rapporteurs :	PIERRE PARADINAS	Professeur titulaire de la chaire Systèmes embarqués au CNAM
	DIDIER DONSEZ	Professeur à l'Université de Grenoble 1
	DANIEL HAGIMONT	Professeur à l'INPT/ENSEEIHT
Examinateur :	CHRISTOPHE BIDAN	Professeur à Supélec Rennes
Directeur :	JEAN-LOUIS LANET	Professeur à l'Université de Limoges

Table des matières

Table des figures

Remerciements

J'exprime ma gratitude à :
— Pierre Paradinas, Professeur titulaire de la chaire Systèmes embarqués au CNAM ;
— Didier Donsez, Professeur à l'Université de Grenoble 1 ;
— Daniel Hagimont, Professeur à l'INPT/ENSEEIHT ;
pour l'honneur qu'ils me font d'avoir accepté la charge d'être rapporteurs et de participer au Jury de ma soutenance d'Habilitation à Diriger les Recherches.

Je voudrais aussi remercier :
— Christophe Bidan, Professeur à Supélec Rennes ;
— Gilles Grimaud, Professeur à l'Université de Lille 1 ;
— Jean-Louis Lanet, Professeur à l'Université de Limoges ;
de les accompagner en tant qu'examinateurs durant cette soutenance.

Le garant de cette habilitation, Jean-Louis Lanet, m'a invité à joindre l'équipe SSD après sa nomination en tant que Professeur à l'université de Limoges en 2007. Pendant 6 ans j'ai travaillé sous ses conseils sur les travaux développés dans ce mémoire. Je lui serai éternellement reconnaissant pour cette expérience extrêmement enrichissante.

Je n'oublie pas l'ensemble des personnes qui ont partagé cette aventure académique, que cela soit dans le travail de recherche mais aussi dans l'enseignement à l'université de Limoges et plus particulièrement dans le Master CRYPTIS. Je pense dans un premier temps à l'ensemble des membres de l'équipe SSD et plus particulièrement à Christophe Clavier qui, outre son amitié, a partagé son expérience sur la cryptanalyse des cartes à puce. L'équipe de cryptographie du département de mathématiques a aussi contribué à cette aventure, notamment avec des personnes comme Thierry Berger, Philippe Gaborit, Marc Rybowicz et Carlos Aguilar Melchior avec lesquelles j'ai collaboré sur des travaux pré-SSD et avec qui je garde une sympathie profonde. Je n'oublie pas non plus l'autre côté de l'informatique à Limoges avec l'ensemble des membres de l'équipe d'image et plus particulièrement Benoit Crespin.

Je remercie les doctorants co-encadrés avec Jean-Louis Lanet : Agnès Noubisi, Ahmadou Séré et Nassima Kamel. Je garde aussi une pensée émue pour Céline Burgod, David Pequegnot et Amaury Gauthier. De plus, je remercie l'ensemble des ingénieurs ayant travaillé sur nos projets dans l'équipe SSD : Hanan Tadmori, Bhagyalekshmy Narayanan Thampi et Matthieu Barreaud. J'accompagne plus particulièrement ces remerciements à Clément Mazin et Jean-Baptiste Machemie qui furent non seulement ingénieurs dans notre équipe mais ont aussi porté la spin-off Arya Security développée sur nos travaux.

De même je remercie les étudiants qui ont effectués sous ma responsabilité ou co-responsabilité des travaux dans notre équipe sous la forme de stages ou de projets. Je pense notamment à Pierrick Buret, Jean Dubreuil, Josselin Dolhen, Patrick Silvera, Anthony Gautrault, Jordan Bouyat, Tiana Razafindralambo, Yorick Lesecque,Josselin Dolhen, Louis Bida, Lonny Brissac, Guillaume Bouffard, Julien Boutet, Emilie Faugeron, Anthony Dessiatnikoff, Damien Arcuset, Eric Linke, Romain Severin, Mamadou L. Balde,

Lylia Tikobaini, Nicolas Tarriol, Jérémy Clément, Silvère Cainaud, Amine Belhociné, Aymerick Savary, Keita Ansoumane et Julie Rispal.

En dehors de mon travail, je voudrais particulièrement remercier du fond du cœur mon épouse et mes enfants pour leur patience durant l'écriture de ce mémoire. Je prolonge ces remerciements à l'ensemble de ma famille et plus particulièrement à mon père, à ma mère et Vinca pour m'avoir encouragé dans mes études.

Enfin, un dernier paragraphe se doit d'évoquer l'ensemble des personnes qui ont relus ce mémoire et plus particulièrement Elisabeth pour avoir survécu au travail de relecture tardif.

Résumé

La Java Card est aujourd'hui le type de cartes à puce le plus déployé dans le milieu bancaire ou dans la téléphonie mobile. Outres la présence de nombreuses contre-mesures physiques pour protéger le microprocesseur contre les attaques externes, la machine virtuelle Java Card possède un ensemble de mécanismes (comme le vérificateur de *bytecode* et le pare-feu) qui, combinés avec le typage du langage Java, offrent des propriétés d'isolation forte des applications (applets) vis-à-vis de l'exécution de la machine virtuelle Java Card.

Mais l'évolution des attaques logicielles par confusion de type et par des moyens physiques a montré des limitations au modèle d'isolation de la machine virtuelle. Dans un premier temps, plusieurs travaux montrent des nouvelles menaces logiques, physiques et hybrides afin de lever des secrets enfouis dans des instances de Java Card en exploitant les applications chargées comme cibles et vecteurs d'attaque.

Par la suite, plusieurs stratégies de contre-mesures sont construites selon deux points de vue. D'une part des protections réactives (contre les attaques en fautes) et proactives (par mise à jour dynamique) sont intégrées dans la machine virtuelle Java Card. D'autre part, des solutions d'analyse de code permettant d'aider le développeur sont évaluées afin de renforcer la sécurité des applets contre des faiblesses de développement ou les exploitations possibles du *bytecode* par des attaques en faute.

Notes

Dans la mesure du possible, les termes anglais ont été évités ou mis en italique dans le cas contraire. Il est très probable que la traduction choisie de certains termes ne reflète pas la traduction officielle, mais elle me semblait plus adaptée. Quant au fait de conserver certains termes anglais, il provient de l'imprécision (fondée sur mon opinion) de trouver un équivalent français de certains termes ambigues ("thread" par exemple). Enfin, il est très probable que des mots anglais soient restés présents malgré ma vigilance.

Afin d'aider le lecteur à séparer mon travail des références citées, tous les identifiants de mes publications citées dans le texte commencent avec 2 ou 3 lettres (par exemple [CIC1] ou [WI3]). La signification de chacune est disponible dans la bibliographie personnelle en page 97.

Chapitre 1

Présentation

Mon travail de doctorat, réalisé à partir septembre 2000 et soutenu en Décembre 2003, s'intéressait à la réduction du coût en énergie de la diffusion de messages dans l'ensemble d'un réseau "ad-hoc" [TH1, RR3, RR2, CIC12, CI9, JIC6, CIC11, JIC7, CIC10, JIC5, CIC9, JIC3]. Par la suite, j'ai été amené à travailler sur la synchronisation entre robots mobiles lors de mon séjour post-doctoral au *Japan Advanced Institute of Science and Technologies* (JAIST) à Ishikawa (Japon) dans l'équipe DDG dirigée par le professeur Xavier Défago [RR1, CIC8, JIC4]. J'ai été ensuite recruté en tant que maître de conférences à l'université de Limoges dans le département d'Informatique où j'ai travaillé avec Carlos Aguilar Melchior sur plusieurs thématiques comme la voix sur IP (VoIP) anonyme (voir section 6.2) ou les problèmes de confiance dans les communications des réseaux ad-hoc (voir section 6.3).

Parallèlement à mes recherches à l'université de Limoges, j'ai eu l'opportunité depuis 2005 de m'occuper de la gestion du Master CRYPTIS spécialisé dans la sécurité des systèmes d'information. Cette formation possède la particularité d'avoir de nombreux intervenants extérieurs offrant une vision récente et pratique de la sécurité réelle des systèmes, des réseaux et des applications. Ma curiosité dans ce domaine et son décalage par rapport à ma vision théorique de la sécurité ont progressivement remplacé mes thématiques de recherche sur les réseaux ad hoc.

En septembre 2007, j'ai intégré à sa création par le professeur Jean-Louis Lanet l'équipe SSD. J'ai alors découvert une recherche appliquée et industrielle sur un sujet de sécurité que sont les cartes à puce Java Card. Jusqu'en septembre 2013, j'ai travaillé sur ce sujet.

Ce document synthétise mes travaux de recherche sur la Java Card effectués au sein de l'équipe SSD du laboratoire XLIM de l'université de Limoges. Un des premiers objectifs de l'équipe SSD est de travailler aussi bien sur les aspects offensifs que défensifs des cartes à puce ; et plus particulièrement sur la Java Card car c'est aujourd'hui le type de carte le plus déployé au monde [73]. À partir de la version 2.1, le déploiement massif de ces cartes dans la téléphonie mobile a installé Java Card comme un standard incontournable utilisé dans de nombreux secteurs d'activité : bancaire, fidélisation, micro-paiement, télévision, authentification, etc.

1.1 Contexte

La carte à puce

Une carte à puce [41] est, comme son nom l'indique, une carte en plastique (norme ISO/IEC 7810 au format hérité des cartes plastiques à pistes magnétiques) dans laquelle est enfouie un circuit intégré. Son aspect extérieur inclut un contacteur standardisé (norme ISO/IEC 7816) mais aussi de plus en plus d'interfaces de communication sans fil comme NFC (*Near Field Communication*).

D'abord simple mémoire, le composant électronique de la carte est remplacé par un microprocesseur. Vu de l'extérieur, ce composant a un rôle simple. Il envoie une réponse à la mise sous tension (*Answer To Reset* ou ATR) afin de s'identifier au lecteur. Puis il traite chaque commande envoyée (*Application Protocol Data Unit* ou APDU) qu'il reçoit et renvoie un code de 2 octets pour indiquer le résultat du traitement (éventuellement précédé des données résultats).

Pour les besoins de flexibilité, la logique câblée monolithique a été remplacée par un microprocesseur contenant un système d'exploitation capable de charger le programme à exécuter. Le système doit donc offrir des abstractions à l'application, assurer la confidentialité et l'intégrité des secrets, gérer les transmissions avec le lecteur et contrôler l'exécution du code.

la technologie Java Card

L'objet d'étude est la technologie Java Card et sa machine virtuelle (*Java Card Virtual Machine* ou JCVM). Cette technologie répond au besoin après émission de la carte du déploiement de plusieurs applications par plusieurs émetteurs au sein de la même puce (on parle alors de cartes multi-applicatives). Les applications sont développées avec un sous-ensemble du langage Java et l'application produite est chargée dans la carte.

Dans la suite de ce document et sauf mention contraire, nous discutons de la version 2.2 de Java Card.

Développement

Le flot de développement d'un applet (une application Java Card) est classique d'un point de vue développeur [101]. Il utilise le langage Java avec une API dédiée pour hériter de la classe `Applet` et surcharger les méthodes servant de points d'entrée à son programme (par exemple la méthode `process` en charge de traiter chaque APDU envoyé à l'application). Les sources sont ensuite compilées avec le compilateur Java. L'avantage technologique induit est donc un temps d'implémentation et de diffusion réduit (par rapport à la nécessité de spécifier et fondre la logique câblée) avec un coût de développement d'application ne nécessitant pas (en théorie) de compétences particulières (sous-ensemble du langage Java et utilisation de différentes API pour le support cryptographique, l'acquisition du PIN de l'utilisateur, la sérialisation et la persistance automatique des instances, ...).

La spécificité du monde Java Card commence ici, avec l'utilisation de l'outil *converter* (fourni avec le kit de développement Java Card) qui regroupe l'ensemble des classes dans un fichier unique (le fichier CAP) en utilisant les informations contenues dans les fichiers export (EXP) afin d'identifier les identifiants numériques associés à chaque classe

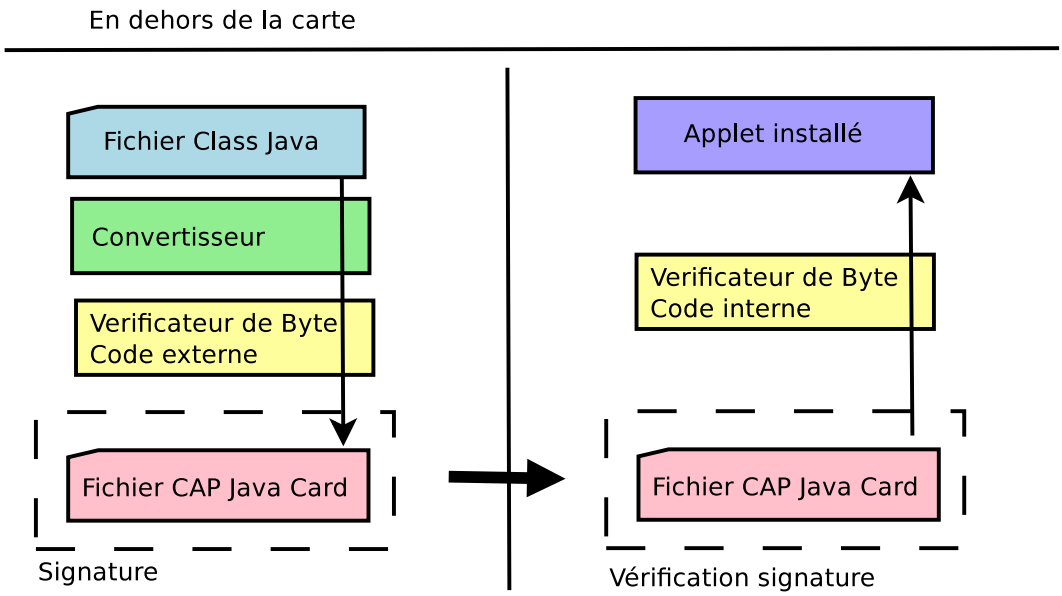


FIGURE 1.1 – Flot de développement Java Card

et *package*. L'application peut alors être chargée, mise à jour ou retirée d'une Java Card après émission de la carte ; On parle alors de *post-issuance*.

La machine virtuelle Java Card est une version minimale de la machine à pile Java : elle ne gère pas de ramasse-miettes ou de *threads*, possède une API limitée et un jeu réduit d'instructions. Une fois chargée et l'édition des liens réalisée, l'applet est exécuté dans un environnement d'exécution (JCRE ou *Java Card Runtime Environment*) gérant le cycle de vie de l'applet et les échanges APDU.

Une fois en main de l'utilisateur final, une carte à puce peut accueillir plusieurs applets de différents fournisseurs en même temps. Il est donc nécessaire d'envoyer une séquence particulière d'APDU à la carte (`select` et `deselect`) afin de sélectionner un applet avant de pouvoir communiquer avec lui.

Écosystème Java Card

Le développement et la diffusion de cartes à puce Java Card s'inscrivent dans un échange de services et de biens entre plusieurs acteurs [83, 85] :

le fondeur producteur du composant matériel : le microcontrôleur intégrant sur une même puce la RAM, ROM, EEPROM et les contre-mesures matérielles décrites dans la section 2.1.

l'encarteur aussi appelé assembleur, achète les composants matériels et développe les piles logicielles associées : système d'exploitation, machine virtuelle, applets fourni avec la plate-forme (administration). De la nécessité de segmenter le marché, l'offre qui est en ressort inclut plusieurs cartes avec plusieurs niveaux de sécurité garantis et une "personnalisation" pour le client (apparence physique, logiciels et secrets installés).

le commandeur ou émetteur en charge de la spécification des besoins, l'exploitation et distribution des cartes. Il peut être amené à personnaliser les cartes pour les besoins d'exploitation.

le prestataire produit des applications Java Card pour l'émetteur. Le fait de dissocier l'émetteur du prestataire permet l'existence de société de services spécialisées dans la production d'applets.

l'utilisateur final ou porteur est le possesseur de la carte à puce.

Sécurité Java Card

Une implémentation d'une JCVM doit garder les objectifs initiaux de la carte à puce (intégrité et confidentialité), c'est-à-dire conserver des secrets et les manipuler de façon sécurisée avec des algorithmes cryptographiques, mais doit aussi garantir l'innocuité des application chargées vis-à-vis de sa machine virtuelle, son système d'exploitation et les autres applications. Pour garantir ces propriétés, Java Card repose sur quelques fondements :

un langage fortement typé la sûreté des types permet de garantir leur usage correct afin d'éviter les manipulations de pointeur par le programmeur.

une vérification du code exécutable : il est possible de vérifier la conformité du fichier de code exécutable (le fichier CAP) par l'usage d'une pièce logicielle appelée le vérificateur de *bytecode* (BCV ou *ByteCode Verifier*) en dehors de la carte ou durant le chargement dans celle-ci.

un chargement sécurisé dans la carte : le protocole GlobalPlatform [87] permet l'authentification, la signature et/ou le chiffrement des applications lors du chargement dans la carte, garantissant ainsi que l'émetteur de l'applet est autorisé et que l'application est transmise de manière sécurisée (intègre et chiffrée).

Un contrôle des accès d'une application : le pare-feu (*firewall*) est une fonction du JCRE en charge de vérifier pendant l'exécution si les accès d'un applet respectent l'isolation par contexte, chaque contexte contenant les instances appartenant au même *package* (deux instances appartenant au même *package* peuvent communiquer sans obstruction par le pare-feu).

Un contrôle des échanges : par l'usage de transaction, le programmeur possède un mécanisme lui garantissant un état cohérent des instances créées même en cas d'arrachage soudain de la carte du lecteur.

Au vu de ces mécanismes de sécurité présents tout au long du cycle de vie de développement, de distribution et d'utilisation des applets, et sous réserve d'une implémentation robuste, Java Card semble être résistant contre de nombreuses attaques logicielles de la part du prestataire de service, de l'émetteur ou du porteur. **C'est dans cette optique que s'inscrit le travail que j'expose dans ce document, c'est-à-dire l'évaluation de cette résistance dans un contexte réel d'utilisation**.

Chapitre 2

Problématiques

"At the end of the day, the goals are simple : safety and security."
— Jodi Rell, *Femme politique américaine*

Le principe de fonctionnement d'une Java Card étant défini, ce chapitre est consacré dans un premier temps à l'étude de la sécurité logique et physique de ces cartes (section 2.1) afin de justifier l'approche prise pendant mes travaux. Afin de concevoir des contre-mesures efficaces, l'importance des menaces sur et de la part des applications Java Card (section 2.2) vis-à-vis des contraintes d'évaluation (section 2.4) et d'intégration (section 2.5) pour l'écosystème industriel (section 2.3) sera caractérisé dans un second temps. L'exposé des verrous technologiques qui découlent de cette approche conclut ce chapitre (section 2.6).

2.1 De la réalité de la sécurité des cartes

Java Card est un standard composé de spécifications publiques. Une carte à puce Java Card est donc une implémentation de cette technologie par un vendeur possédant une licence pour l'exploitation de la "*Java Card Technology*". Il existe donc :

- une ou plusieurs implémentations de la machine virtuelle Java Card, chacune avec différentes versions et modifications ;
- réparties à travers plusieurs vendeurs de cartes, chacun proposant une multitude de modèles de cartes.

Même si les spécifications sont publiques, il existe certaines difficultés dans le développement d'une implémentation d'une carte à puce Java Card. En effet, les documents ne contiennent qu'une description fonctionnelle :

- de la machine virtuelle, de son jeu d'instructions et du format des fichiers (CAP et EXP) disponibles dans le document *Virtual Machine Specification* [76] ;
- du cycle de vie des applets et des composants du JCRE dans le document *Runtime Environment Specification* [74] ;
- des méthodes de la bibliothèque Java Card dans le document *Application Programming Interface* [70].

La valeur ajoutée des industriels encarteurs de Java Card par rapport à ces spécifications est la capacité de sécuriser la machine virtuelle contre des attaques logiques (voir section 2.1) et physiques (voir section 2.1) qui sont en dehors des documents de référence fournis par Sun/Oracle.

L'aspect "boîte noire" de ces composants est donc non seulement la fonction première de protection des opérations et données, mais aussi un secret industriel pour éviter d'identifier par des attaquants ou des concurrents les contre-mesures disponibles sur les cartes.

Dans l'impossibilité de présenter un détail des protections aux clients, les encarteurs appuient les arguments commerciaux de sécurité des cartes Java Card principalement sur une reconnaissance de l'évaluation par des organisme d'évaluation (CESTI) afin d'atteindre des niveaux de protection relatifs à des normes telles que les critères communs (EAL).

Une autre raison de l'opacité des mécanismes internes d'une Java Card est de valoriser les possibles services associés, tels que les outils de développement ou encore la certification d'application. En maîtrisant le cycle de vie complet des supports d'exécution et des applications chargés sur la carte, le but est d'enfermer les clients dans des processus ou des outils propriétaires afin de les garder captif d'un vendeur malgré l'aspect "write once, run everywhere" de Java (Card).

Il est donc très difficile de connaître la réalité (*i.e.* les mécanismes internes) de la sécurité des cartes Java Card en circulation. Construire une recherche autour de cette technologie nécessite alors soit de s'aligner sur des laboratoires d'évaluation (CESTI) avec un investissement matériel et humain important, soit de trouver des stratégies originales d'attaques.

> Afin de pouvoir proposer des solutions réalistes vis-à-vis des possibilités d'un attaquant, notre équipe s'est orientée dans le développement d'attaques originales afin d'évaluer la sécurité réelle des implémentations de Java Card pour proposer des contre-mesures adaptées. Il faut donc soit évaluer la sécurité réelle des cartes, soit avoir un modèle réaliste des attaques, et en déduire alors des contre-mesures.

Dans la suite de cette section est présenté le positionnement par rapport aux travaux existants sur l'évaluation de sécurité de carte à puce afin de justifier les directions prises par mes recherches.

Des attaques matérielles

Une première classe de menaces concerne l'atteinte à la confidentialité des secrets dans le cas où la carte est en possession de l'attaquant. Ce scénario est réaliste dans le cas de cartes à puce pour la télévision satellite, où la compromission d'une seule carte a un impact très important pour l'opérateur.

Les approches pour étudier physiquement les cartes à puce se répartissent en plusieurs catégories [16] : inspection, observation et perturbation.

Inspection

Cette approche représente une classe d'attaques invasives car elle modifie (voire détruit) le matériel de manière irréversible. Après extraction du microprocesseur de son support plastique et l'utilisation d'acide pour enlever la couche de résine, le microprocesseur nu peut-être étudié à l'aide d'un microscope ou par la pose de micro-sondes sur les bus.

Mais le niveau d'expertise pour monter de telles attaques est important car elles nécessitent des connaissances et un équipement dans le domaine de la chimie, de l'imagerie et/ou de la micro-électronique. De plus, il existe déjà un ensemble de contre-mesures comme des cages de protection emballant le microprocesseur (effaçant les secrets en cas d'altération), des détecteurs de lumière (en cas d'extraction physique de la puce), le chiffrement de la mémoire / bus, ou l'offuscation des pistes du circuit électronique.

Cette catégorie d'attaques est en dehors de nos compétences car elle nécessite une lourde expertise sur l'aspect matériel et humain. De plus, notre expertise est principalement orientée sur le logiciel, donc il est difficile d'apporter des solutions à ce niveau lors d'écoute par des micro-sondes.

Observation

L'observation concerne l'écoute des grandeurs physiques du microprocesseur pendant l'exécution (on parle alors de canaux cachés) : le temps de réponse [111], la consommation électrique [108] ou les émissions électro-magnétiques [99].

Il existe une importante littérature scientifique à ce sujet en grande majorité consacrée à la cryptanalyse. En effet, par l'étude des variations des grandeurs physiques, il est possible de déduire selon un modèle du système différents canaux cachés représentant des fuites d'information pouvant mener à la découverte des secrets enfouis. Par exemple, lors de l'écoute de la consommation de courant, des méthodes comme l'analyse de la consommation simple de courant (*Simple Power Analysis* ou *SPA*) [86, 104] ou de la consommation différentielle de courant (*Differential Power Analysis* ou *DPA*) [108] permettent de déduire les opérations effectuées par le microprocesseur. D'autres canaux cachés existent, comme découvrir le nombre d'étapes exécutées par un algorithme cryptographique [102], l'exécution ou non de branches d'un test conditionnel, les défauts de cache [63], les accès en lecture ou en écriture à la mémoire, ou encore le comportement de l'algorithme cryptographique lors de la soumission d'entrées volontairement faussées.

En dehors de la cryptanalyse, l'observation de grandeurs physiques peut mener au rétro-ingénierie à partir de la connaissance des spécifications du micro-processeur. Ainsi il est possible d'identifier le code binaire exécuté au niveau du microprocesseur [18] mais aussi, dans le cas de la Java Card, d'identifier le *bytecode* exécuté à partir des motifs de consommation de courant par des instructions élémentaires du processeur [60].

En terme de contre-mesures, il n'est pas possible d'avoir de modèles actifs de mécanismes de défense car l'attaquant ne fait qu'observer. Les solutions sont donc de lisser le résultat (pour le rendre uniforme) ou de diminuer le ratio entre le signal informatif et le bruit.

La première méthode est l'équilibrage (en consommation ou en temps de calculs) afin d'obtenir un signal constant quelque soit les données manipulées. Une méthode est de traiter toutes opérations en temps constant, comme par exemple de tester les 4 chiffres d'un code PIN même en cas d'échec du test d'un chiffre. Sur les algorithmes cryptographique, plusieurs propositions pour exécuter les algorithmes en un temps fini ont été proposées pour AES et DES [96].

Une seconde méthode est l'ajout de bruit pour diminuer le ratio signal/bruit, par l'ajout d'instructions, la redistribution de manière aléatoire des groupes d'instructions au niveau de la compilation, ou encore l'augmentation artificielle de la gigue de l'horloge.

A la lecture des ces attaques et contre-mesures par observation, il apparaît la nécessité d'un bagage important dans l'analyse du signal, qui n'était pas un de nos domaines d'expertise à l'époque de la création de l'équipe et de ses premiers travaux (elle l'est depuis grâce à l'arrivée du professeur Christophe Clavier). De plus, les contre-mesures semblent aujourd'hui répondre efficacement au menaces posées (sur des cartes haut de gamme subissant un niveau de certification important).

Perturbation

Une dernière approche consiste à perturber le fonctionnement de la carte par la manipulation temporaire ou permanente des grandeurs physiques (aussi appelée "glitch") comme l'alimentation délivrée par le lecteur de carte [23], mais aussi l'exposition à un champs électromagnétique [99] ou à la lumière [90].

Ces attaques sont difficiles à mettre en œuvre car elles demandent un laboratoire physique dédié. De plus, il existe de nombreuses contre-mesures [62] depuis les premières démonstrations de ces attaques : détecteur de lumière ou capteur de valeurs physiques non conformes dans les entrées par exemple.

Il existe néanmoins une attaque considérée comme réaliste et encore valide : l'injection de fautes par laser [62]. Cette spécialisation des attaques lumineuses exploite la sensibilité des circuits électriques à la lumière pour engendrer des fautes. Mais par rapport aux attaques par flash, le faisceau est directionnel et peut-être concentré sur une portion réduite du microprocesseur ciblé (de l'ordre de quelques μm^2, proche de la taille d'un mot mémoire). La nécessité de taille réduite du microprocesseur ne permet pas de disposer suffisamment de détecteurs de lumière pour couvrir l'ensemble de la surface du silicium, laissant d'autant plus de point d'entrée à un attaquant.

Les moyens matériels et humains pour monter cette classe d'attaque est encore une fois en dehors des ambitions de notre équipe. Par contre, contrairement aux deux autres classes d'attaques physiques, il est possible, à partir d'un modèle d'attaque en faute, d'évaluer l'impact sur du code embarqué dans la carte, et plus précisément le *bytecode* des applications installées, le code natif de la machine virtuelle étant indisponible.

Des attaques logicielles

Les attaques logicielles pour une carte à puce se situent au niveau de ses entrées / sorties. Le système d'exploitation et la machine virtuelle doivent disposer de méthode robustes de traitement des entrées APDU, mais aussi refuser des opérations non conformes, voire des états non attendus (comme l'arrachage de la carte du lecteur durant l'exécution d'une opération). Les protections et contre-mesures font partie de l'expérience des encarteurs et l'on peut attendre un très haut niveau de robustesse.

Dans le cas d'une Java Card, un autre vecteur d'entrée est l'installation d'un applet malicieux (dont le contenu est sous la maîtrise de l'attaquant), et ainsi exploiter de potentielles faiblesses de la carte lors de l'exécution de cet applet. Cette idée est intéressante car la disponibilité de cartes développeurs pour des prix modiques sur Internet permet à un attaquant de posséder physiquement la carte et les clés de chargement.

Depuis 2004, Les principaux travaux d'applets malicieux proviennent de Erik Poll, Engelbert Hubbers et Wojciech Mostowski. Au cours de la formalisation du mécanisme de transactions de la machine Java Card [79, 66], ils ont découvert une ambiguité dans les spécifications Java Card lors de l'usage des méthodes `Util.arrayCopyNonAtomic` et `Util.FillNonAtomic` [72] lors de transactions. Les auteurs démontrent sur deux cartes du marché que les objets ne sont pas tous détruits correctement, notamment selon que que l'objet est persistant ou pas. Ce comportement peut amener à des accès non autorisés car un objet est considéré comme détruit suite à l'échec de la transaction mais sa référence reste toujours valable. À la suite de ces travaux, la spécification 2.2 de Java Card précise que l'usage de deux méthodes sur un tableau durant une transaction n'est pas prédictible à la suite d'une remise à zéro ou d'un arrachage.

Une autre approche d'attaque des mêmes auteurs concerne le cas du mécanisme de partage [57]. Sur au moins une carte, Il est possible de faire des appels de méthode à une instance n'appartenant pas à son contexte si cette instance est marquée comme partageable. Afin de connaître les fonctions accessibles, le programmeur en charge de la création de l'applet partage l'interface implémentée par l'instance accessible. Un attaquant peut alors modifier son applet afin que le *bytecode* exécutant un appel de méthode sur l'objet distant ne respecte pas le typage des paramètres du prototype provenant de l'interface. L'applet obtenu est ainsi toujours valide (il est jugé correct par le BCV) mais il n'y a aucune vérification par la machine virtuelle de la correspondance du type de l'interface entre l'appelant et l'appelé. Un exemple est d'appliquer une confusion de type pour changer la déclaration dans l'interface de partage d'un tableau de *byte* par un tableau de *short* afin d'accéder à des portions mémoires inaccessibles de l'applet appelé.

Au regard de ces résultats, les auteurs ont montré des faiblesses limitées dans certaines implémentations de la JCVM et du JCRE en présence d'un BCV qui ne remettent pas en cause le modèle de sécurité de Java Card, à la suite de deux inconvénients majeurs :

— Le meilleur résultat obtenu est l'accès à une portion mémoire limitée de l'EEPROM contiguë au tableau sur lequel s'applique la confusion de type ;
— De nombreuses cartes ne proposent pas le support des mécanismes de partage, des normes de certification d'applets Java Card comme AFSCM (voir section 5.3) interdisant leur utilisation ;

Le travail le plus intéressant de ces auteurs est la manipulation de fichiers CAP sur des cartes ne possédant pas de BCV afin de mener des attaques de confusion de type [52] : changement du type d'élément contenu dans un tableau (*byte as short array*) ou accès à un objet vu comme un type tableau (*object as an array*). Cette dernière possibilité ouvre le champs à d'autres attaques car il est possible alors de forcer la machine à traiter les objets comme des tableaux afin d'accéder à la valeur numérique interne des références contenues dans l'objet.

Les attaques logicielles proposées par Erik Poll *et al.* permettent d'observer la représentation interne des structures dynamiques pendant l'exécution de l'applet (valeur des références ou entête des objets). Mais la confusion de type ne s'applique qu'à un sous-ensemble des instances et ne donne qu'une vision partielle de la représentation en mémoire de l'applet. De plus, le pare-feu empêche tout accès externes à l'applet en cours d'exécution, isolant ainsi les fuites d'information à l'applet malicieux.

Erik Poll *et al.* ont démontré la possibilité d'utiliser l'application comme vecteur d'attaque pour capturer des états internes de la carte. Notre approche sera de poursuivre ces travaux en s'intéressant à dépasser l'isolation par le pare-feu.

2.2 De l'attaque des et par des applications

Par la méconnaissance des mécanismes internes et l'incapacité à mener des attaques physiques réelles, la Java Card en tant que support physique et support d'exécution (JCVM) n'est pas une cible à privilégier. Mes travaux ont privilégié les applications (applets) chargées sur la carte selon deux aspects :

l'application comme cible de l'attaque

Un encarteur a comme métier de protéger la plate-forme, et non les applications qui y sont chargées car leur sécurité est à la charge du développeur (prestataire de service). Mais paradoxalement, le manque de connaissance des développeurs sur les aspects sécurité et le culture du secret dans l'industrie de la carte (par la présence d'une activité importante de certifications fonctionnant sur le secret et le savoir-faire) ne permettent pas le déploiement ouvert des bonnes pratiques de la sécurité dans la carte. De plus, l'impossibilité de connaître les mécanismes internes de la carte et les contre-mesures disponibles rendent impossible le développement efficace d'applications sécurisées, notamment contre les attaques physiques.

l'application comme vecteur d'attaque

L'accès à des cartes développeurs pour un coût faible offre à un attaquant la possibilité d'évaluer la surface d'attaque de la machine virtuelle et des composants associés (mécanisme de gestion du cycle de vie des applets ou API). Normalement la machine virtuelle doit offrir une résistance importante (c'est le métier de l'encarteur et sa réputation) mais la démonstration des attaques de Poll *et al* montrent qu'il y a un intérêt à poursuivre ce vecteur d'attaque.

Ces choix évitent de se confronter directement à l'aspect boîte noire de ses composants par la disponibilité publique du format des applications et des protocoles de communication, inclus dans les différents SDK de Java Card ou dans la standardisation par des organismes comme GlobalPlatform. Mais je prends alors l'hypothèse que le format exécutable reste consistant lors du chargement dans la carte, hypothèse réaliste vis-à-vis des faibles ressources de la carte.

2.3 De l'importance des relations industrielles

Mon travail de recherche est principalement appliqué et s'appuie sur un réseau de partenaires industriels. Ce choix influence mes orientations de plusieurs façons.

Dans de nombreux travaux, un biais possible est que le modèle d'attaque soit spécifiquement ajusté afin de mettre en valeur la solution proposée. Mais il est en général difficile de trouver des justifications de ce modèle et notamment de sa réalité pratique. Dans le cas de Java Card, on peut citer le cas de la caractérisation du modèle de faute sur le matériel. Il existe une littérature très importante dans le domaine, notamment cryptographique. Or le pouvoir de l'attaquant est défini vis-à-vis d'un modèle de faute, c'est-à-dire de la capacité d'un attaquant de modifier le comportement de la puce par altération de son environnement matériel. Mais cette donnée n'est en général pas publique et le travail scientifique d'évaluation cryptographique s'arrête à proposer des attaques et contre-mesures en fonction d'un modèle de faute. Ce problème se répète dans la maîtrise

d'une charge logicielle par un attaquant vis-à-vis de la disponibilité de certaines cartes développeur dans le cadre des attaques EMAN (voir section 3.2).

Un modèle d'attaquant fourni par les partenaires (basées sur leur connaissance actuelle, pas forcément publique, et leur capacité d'anticipation) me permet de proposer des solutions adaptées dans ce cadre.

Deuxièmement, un de mes autres objectifs est de mettre à disposition des outils à bas coût, pertinents et réutilisables par nos partenaires industriels (voire pour le grand public), afin de consolider nos relations en montrant notre savoir et en permettant le transfert technologique.

Ce choix implique des choix et orientations dans les projets car la main d'œuvre pour le développement est en général limitée (doctorants, ingénieurs ou projet étudiants) d'où un choix défini d'orientations dans mes stratégies de développement dans l'équipe et le design de ces outils, notamment dans les capacités d'évolution et de qualité suffisante pour faciliter la reprise et le suivi.

2.4 De l'évaluation de nos propositions de contre-mesures

Un de mes objectifs est d'avoir des solutions réalistes vis-à-vis des contraintes industrielles. La règle qui est prise dans mes travaux est celle des "5%" ; c'est-à-dire que l'ajout d'une solution de sécurité doit entraîner un surcoût d'exécution dans la carte au plus de 5% par rapport à une exécution sécurisée. Cette contrainte est prise vis-à-vis des partenaires industriels car toute solution ayant un coût plus élevé sera rejetée.

Ce surcoût est appliqué au chargement d'un applet (taille du transfert et traitement additionnel après installation) et pendant l'exécution dans la carte, c'est à dire au niveau du système d'exploitation ou de la machine virtuelle Java Card (par exemple par l'augmentation du temps de traitement au niveau *bytecode* dans le cas de sa réécriture).

Au niveau mémoire, il faut bien distinguer 3 cas : celui de la RAM, la ROM et l'EEPROM. La distinction est importante car chacune a un impact très particulier sur le coût de la carte. A commencer par la RAM dont le total est limitée à une taille relativement petite (2Ko de RAM pour une carte embarquant une JCVM 2.1), donc l'ajout d'information dans cette partie est réellement critique. Le second cas est l'EEPROM, lorsque l'application contenue dans fichier CAP est modifiée ou si des méta-informations sont ajoutées parallèlement à l'application (en sachant que les temps d'écriture en EEPROM sont considérablement plus élevés que dans les autres types de mémoire). Enfin, en ROM, ce sont les applications *romisées* et la taille du code exécutable natif qui sont prises en compte.

En dehors de la carte, l'ajout ou la modification du processus durant le développement, la compilation, la conversion (*converter*) et plus généralement avant le chargement du fichier CAP doivent être quant à eux quantifiés en tant que surcoût en terme d'organisation. Par exemple, l'utilisation d'outils automatiques d'analyse statique peut ajouter

du temps dans le processus du développement mais peut apporter en contrepartie un réel bénéfice en terme de sécurité pour les acteurs concernés.

Enfin, il reste aussi l'efficacité ; celle-ci est mesurée en fonction du taux de réussite contre des modèles d'attaquants d'une machine virtuelle protégée vis-à-vis d'une machine virtuelle non protégée.

2.5 De l'intégration dans l'écosystème Java Card

Les acteurs de l'écosystème Java Card (décrits dans la section 1.1) possèdent théoriquement un rôle précis pour répondre à un besoin spécifique du marché. Mais la volonté de garder captif des clients de la part des encarteurs peut amener à un certain mélange des genres.

Par exemple, l'encarteur doit délivrer une carte à puce dans son support plastique et disposant d'une JCVM répondant au besoin de l'intégrateur (c'est-à-dire une Java Card prête à être personnalisée), l'intégrateur prend alors en charge la personnalisation et la distribution du support au client final. Mais dans la réalité, il peut exister une association entre le produit délivré et des outils propriétaires (comme le développement, le chargement d'applet ou le support de nouvelles fonctionnalités non couvertes par les spécifications Java Card) de la part de l'encarteur afin de conserver une maîtrise du marché.

De même, les acteurs en charge de la certification se positionnent après le développement par le prestataire de service pour certifier une application avant l'intégration, ou peuvent encore considérer un produit fini (intégrant l'ensemble des applications) afin de certifier l'ensemble logiciel et matériel. Mais la connaissance des mécanismes internes de la carte et la culture du secret peut réduire la portée de la certification du produit, sans compter le rapprochement entre les entités de certification et les encarteurs.

Cette concentration a aussi le défaut d'affaiblir les acteurs post-émission, car le manque d'information et de culture sur la sécurisation d'applets (par l'absence d'information sur les mécanismes internes par exemple) entraîne une réduction de la sécurité par les prestataires de services qui ne peut-être compensée que par un coût élevé de certification.

> La concentration des moyens au niveau des encarteurs est en contradiction avec un écosystème Java Card ouvert mais aussi pour les acteurs de la recherche, qui se retrouvent dépendants d'outils et de procédures imposées. Il est donc important que les objectifs de ma recherche permettent d'utiliser voire de fournir des outils et des processus compatibles avec le flot classique de développement Java Card.

De la disponibilité des outils

Une des caractéristiques du milieu de la Java Card est le contraste entre la disponibilité des guides de référence et l'opacité / indisponibilité des outils, qui se classent dans 3 catégories :

— La disponibilité sous forme binaire uniquement mais gratuite (avec des restrictions dans l'usage par la licence associée) : le CAP *converter* et le compilateur *javac* par exemple ;

— La disponibilité sous des conditions restrictives et/ou sans code, pour un coût d'acquisition et de maintenance élevé : les IDE des principaux encarteurs par exemple ;
— L'indisponibilité totale en dehors de quelques partenaires : les outils d'évaluation par les CESTI ou les encarteurs, les implémentations de la machine Java Card (voire l'implémentation de référence) et les tests de compatibilité Java Card fournis par Sun/Oracle.

La disponibilité partielle des outils pose problème pour mes expérimentations pour plusieurs raisons. Premièrement, la confiance dans un outil binaire ne nous permet pas de vérifier son comportement comme cohérent. Un exemple est la faille sur le compilateur *java* et le CAP *converter* découverte par Faugeron *et al.* [37], qui a montré l'intérêt d'inclure la certification de ces composants dans le processus de Common Criteria [38]. Deuxièmement, les outils ne délivrent pas le moyen de connaître une représentation interne intermédiaire ou de la manipuler, ce manque de flexibilité limitant la granularité des positions dans le flot de développement et de déploiement où greffer nos recherches. Enfin, certains outils ne sont tout simplement pas disponibles et doivent être recrées de toutes pièces. Cela est particulièrement le cas pour tous les outils d'évaluation de sécurité.

Un autre objectif est de permettre de disposer d'outils ouverts pour ma recherche mais aussi pour les développeurs (prestataires) et les intégrateurs. En effet, ceux-ci ont peu d'expérience sur le développement d'applets sécurisés, et ont recours à des organismes de certification pour les valider.

De l'intégration dans le flot de développement

Un autre objectif de mes travaux est de développer des outils et des méthodologies permettant de sécuriser les charges logicielles invitées. En effet, la sécurisation de la machine virtuelle est un travail dont la responsabilité incombe à la partie contrôlant le matériel (le fondeur et l'encarteur).

Je montre d'ailleurs dans mes travaux que des faiblesses de sécurité reposent sur les entités en charge du développement des chargés invitées. Il faut donc proposer des solutions de sécurisation pour ces développeurs. Mais la poursuite de la réduction des coûts entraîne le développement des charges logicielles invitées sur des budgets de plus en plus limitées, d'où la nécessité de développer des solutions :

compatibles : elles ne doivent pas remettre en question le flot de développement Java Card ;

incrémentales : les solutions doivent être intégrables à bas coût dans le cycle de développement. Pour cela, il est recommandé que la méthodologie insère des outils et des méthodes au fur et à mesure du développement ;

automatiques : ou plus exactement limitant l'expertise humaine (spécialisée donc chère) et/ou le surcoût de travail de la part du développeur (en terme d'acquisition de connaissance, de temps de développement et de maintenance de la charge logicielle).

Mais la position des développeurs des charges logicielles invitées est au-dessus de la machine virtuelle, donc présentant une vision réduite du système. Or c'est sa connaissance des possibilités qui permet de protéger efficacement une charge logicielle invitée. Mais pour des raisons techniques, de secret ou pour conserver un avantage commercial, les encarteurs

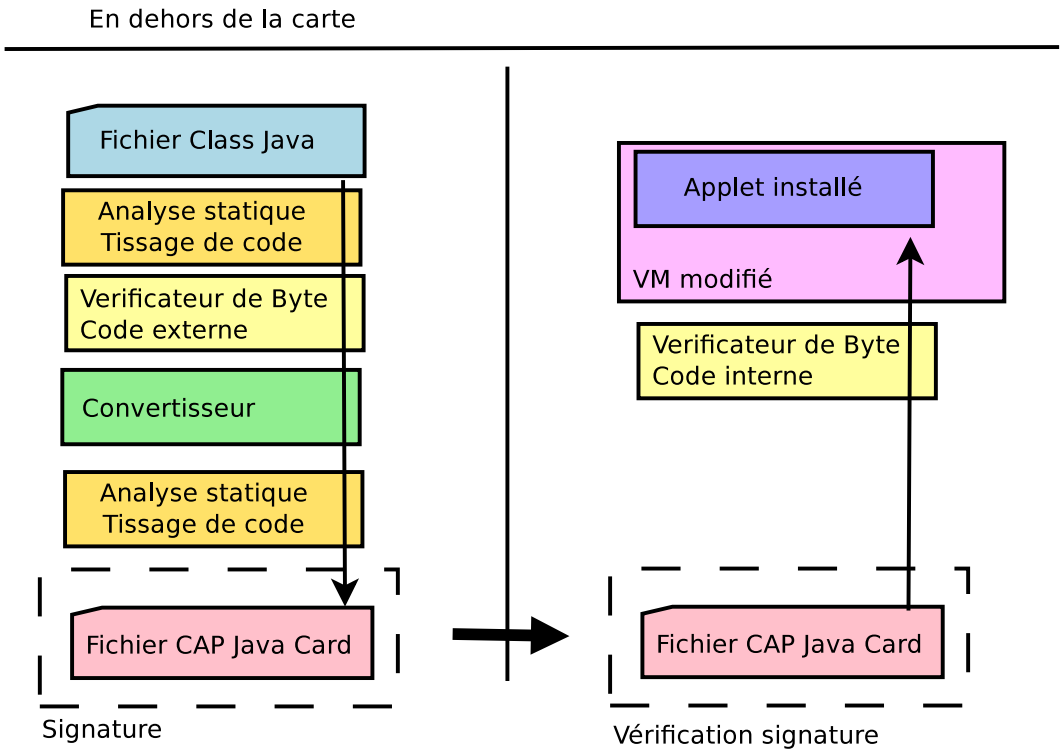


FIGURE 2.1 – Flot de développement Java Card

ne fournissent pas ces informations. Il y a donc nécessité de proposer des solutions et/ou méthodologies de sécurisation des charges logicielles invitées :

— protégeant les savoirs et les avantages commerciaux des développeurs de machine virtuelle
— et indépendantes de l'implémentation de la machine virtuelle

Ainsi, les résultats de mes travaux doivent-ils s'intégrer dans le flot de développement en utilisant plusieurs stratégies (voir figure 2.1) :

analyse statique : au niveau source ou du fichier CAP

tissage du code : au niveau source ou du fichier CAP afin d'ajouter des contre-mesures dans l'application sans remettre le fonctionnement en question

extension des fonctionnalités de la JCVM : sans remettre en question le format binaire des fichiers CAP

2.6 Exposé des verrous scientifiques

Mon approche se concentrant sur l'importance d'évaluer la sécurité au regard du rôle des applications en respectant l'écosystème Java Card, elle amène les 3 principaux verrous scientifiques étudiés dans ce document :

Quels sont les impacts d'attaques matérielles sur les applications ?

Par l'absence de visibilité sur la plate-forme d'exécution, la question est d'étudier l'influence possible d'une faute matérielle (en fonction d'un modèle donné) sur notre application (dont le format est connu).

Quels sont les impacts d'attaques logicielles sur la VM par l'intermédiaire des applications ?

Par la possibilité de maîtriser le format binaire d'une application, la question est d'évaluer les possibilités de modifications malicieuses contre la plate-forme d'exécution afin d'outrepasser l'isolation délivrée par la JCVM et le pare-feu.

Contres les attaques exposées, quelles sont les stratégies de contre-mesures à prendre ?

Au regard des faiblesses identifiées dans les deux précédents points, la question est de proposer des solutions adaptées qui respectent le flot de développement Java Card dans une approche réactive (contre-défense naturelle) et proactive (mise à jour).

2.7 Programme

Mes travaux s'intéressent à lever les 3 verrous présentés dans la section 2.6. Le détail du programme de recherche est de décomposer les trois verrous en trois axes constitués de 7 thèmes. Un détail de cette décomposition associé aux ressources est disponible dans le chapitre 9.

Axe 1 : De l'attaque de la JCVM par des applications malicieuses

Dans un premier temps, je présente dans le chapitre 3 les différents travaux amenant à questionner la sécurité d'une carte Java Card au niveau logiciel :

Thème 1 Évaluation de la résistance de l'implémentation d'un protocole de communication

Thème 2 Évaluation de la résistance de l'isolation de la machine virtuelle contre des application malicieuse

Axe 2 : Défense de la machine virtuelle

Le deuxième axe décrit dans le chapitre 4 concerne les contres-mesures dans la machine virtuelle contre les attaques introduites dans le précédent axe :

Thème 3 Renforcement de la machine virtuelle contre les attaques matérielles sur le *bytecode*

Thème 4 Mise à jour dynamique d'applet

Axe 3 : Défense des applications Java Card par analyse statique

Dans cet axe, présenté dans le chapitre 5, je m'intéresse à l'usage d'outil d'analyse de bytecode afin de pouvoir renforcer la sécurité des applications, et plus particulièrement en montrant les différentes approches possibles qui se sont présentées dans le cadre de mon programme de recherche.

Thème 5 Analyse du *bytecode* d'applets pour le suivi de recommandations

Thème 6 Renforcement d'applets webs contre les attaques XSS

Thème 7 Analyse de l'impact d'attaques matérielles en faute sur le *bytecode* d'une application

Chapitre 3

De l'attaque de la JCVM par les applications

> *"Nobody expects the Spanish Inquisition ! Our chief weapon is surprise...surprise and fear...fear and surprise...our two weapons are fear and surprise...and ruthless efficiency...Our three weapons are fear, surprise, and ruthless efficiency...and an almost fanatical devotion to the Pope...Our four...no...amongst our weapons.... amongst our weaponry...are such elements as fear, surprise.... I'll come in again."*
>
> — Michael Palin as Cardinal Ximénez, *Monty Python Flying Circus, Series 2, Episode 2*

Le travail présenté dans ce document est construit sur l'idée d'évaluation de la sécurité d'instances de Java Card afin de justifier les solutions de contre-mesures proposées par la suite. Ce premier axe s'intéresse à l'évaluation de la sécurité des Java Card d'un point de vue logiciel selon deux approches. Dans la section.3.1, le modèle d'attaquant considéré est la maîtrise du canal de communication (émission et réception des APDU d'une carte Java Card). Dans la section 3.2, le modèle d'attaquant est étendu avec la capacité de charger des applications sous son contrôle afin d'induire des erreurs lors du traitement interne du code de l'application.

3.1 Évaluation des protocoles de communication

Dans un premier temps, mes travaux s'intéressent à l'évaluation de la sécurité et de la conformité de l'implémentation de protocoles de communication sur des cartes Java Card supportant un serveur web, avec la thèse de Nassima Kamel [ST2]. Notre équipe a eu accès dans le cadre d'un partenariat industriel à différentes cartes à puce de type Java Card Web Server, c'est-à-dire incluant une machine virtuelle JCVM (version 2.2.1 ou 2.2.2) et un serveur web [106]. Les cartes supportent le chargement d'applets web développés sur une API dérivée des applets web JEE, et incluent deux protocoles de communication afin de permettre l'accès à ces applications webs (voir figure 3.1) :

Bearer Independant Protocol (BIP) [35] est le protocole passerelle entre la carte et un correspondant adressable par TCP/IP. Le rôle de BIP dans la carte est d'encapsuler les paquets HTTP dans des trames APDU, et une pile logicielle à l'extérieur de la carte (par exemple le téléphone portable dans lequel est enfiché la carte) s'occupe de convertir les trames BIP en paquets TCP/IP.

HyperText Transfert Protocol (HTTP) [109] est le protocole applicatif classique du *World Wide Web*. Il est sans état et permet de demander au serveur une ressource

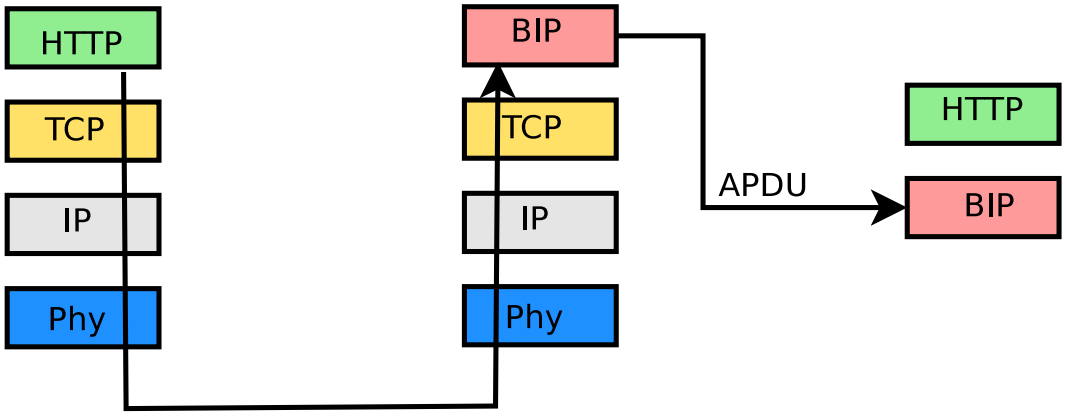


FIGURE 3.1 – Support HTTP et BIP

identifiée sous forme d'URI dans une requête au format texte contenant une action (GET, POST,...) complétée par des entêtes.

Le principal problème est l'aspect boîte noire de ces deux implémentations. Le code source ou le binaire ne sont pas disponibles. Une recherche de vulnérabilités par analyse statique des sources ou du binaire n'est donc pas faisable.

La carte est par sa nature-même avare de fuites d'informations durant son exécution. A priori, il est donc difficile de connaître les états internes de la carte ou les chemins d'exécution suivis dans le but de maximiser la couverture des tests. De plus, les attaques matérielles par écoute des grandeurs physiques ne sont pas dans notre programme de recherche. Il reste donc l'évaluation au niveau logiciel des données encapsulées dans des APDU [1]. Ainsi, l'évaluateur envoie des stimuli en entrée de la carte cible et analyse les réponses afin d'en déduire des informations sur les possibles faiblesses.

L'étude de la sécurité de l'implémentation d'un protocole est réalisable de plusieurs manières [94]. Il faut d'abord que le référentiel du protocole soit complet et cohérent en terme de sécurité, par exemple en le modélisant et en le vérifiant par des méthodes de *model checking*. Une approche de la communauté des tests est alors de construire un modèle de ce protocole et de générer l'ensemble des tests vérifiant son implémentation.

Mais mes travaux ne se sont pas orientés vers ces solutions pour plusieurs raisons. Premièrement, le travail d'évaluation est réalisé par un ingénieur sur un contrat court et/ou par des étudiants dans le cadre de leur projet de Master 1. Il est donc difficile d'envisager de les former sur les outils de modélisation et génération de tests (sans compter qu'il n'existe pas d'expérience dans l'équipe sur ces outils), notamment avec la difficulté de trouver une formalisation du protocole évitant l'explosion combinatoire des états (par exemple avec une représentation symbolique intermédiaire). Deuxièmement, nous considérons une évaluation rapide de l'implémentation, ce qui est incompatible avec la nécessité de déployer une approche de type *model checking*, notamment par l'aspect infini des formats de HTTP par exemple.

Le fuzzing est une alternative demandant un coût inférieur en terme de modélisation et d'expérience nécessaire de la part de l'évaluateur. La cible étudiée reçoit en entrée des séquences altérées par rapport au format attendu selon deux approches [56] : soit par le

1. La couche APDU peut être qualifiée comme très sécurisée car en tant que protocole bas-niveau de communication d'une carte à puce, elle bénéficie d'une longue expérience de la part des développeurs de système d'exploitation pour carte.

rejeu de séquences enregistrées préalablement mais modifiées (par *bit-flipping*, mutation/-croisement, etc.), soit par génération à partir d'une grammaire des entrées de séquences modifiées par différentes politiques selon le type du champs. L'absence de réponses, la fuite d'informations dans les réponses ou encore le crash de la cible peut alors dénoter un impact sur la stabilité du système. Cette méthode a été appliquée sur de nombreuses cibles, quelques soient les entrées (fichiers, trames réseaux) et fait partie de l'arsenal de base des évaluateurs sécurité [59]. Il est aussi efficace sur du format binaire [11] que sur des protocoles textes qui possèdent des classes d'attaques particulières, par exemple sur l'encodage des caractères.

Par rapport à une modélisation complète du système, le fuzzing est une méthode incrémentale car son efficacité dépend du temps affecté à l'évaluateur (une mission d'évaluation par "fuzzing" est d'ailleurs souvent bornée en durée). Par exemple la première approche par rejeu de séquences altérées à un coût très faible en terme de déploiement (des traces de communication sont enregistrées puis rejouer avec des modifications). De son côté, la méthode de modélisation par grammaire peut se limiter par exemple à un sous-ensemble du protocole, certains types de paquets et/ou sans considérer les états du système (contrairement au *model checking*).

Plus généralement, la méthode "traditionnelle" de fuzzing ne cherche pas à vérifier la conformité d'un protocole cible (pouvant nécessiter une compréhension complète de sa spécification) mais génère de manière pseudo-aléatoire des séquences d'entrées. Selon la méthode et l'expérience de l'évaluateur, cette génération peut effectuer de simple échanges / substitutions jusqu'à privilégier des séquences ayant un haut potentiel de perturbation de la cible car visant des motifs connus pour être source de faiblesses [20].

L'expérimentation développée dans ce cadre a utilisé la version 2 de Peach [2], un logiciel open-source de fuzzing par grammaire. Il possède une bibliothèque de comportements pouvant mener à des cas intéressants à tester pour chaque champs défini dans la grammaire du protocole cible (valeurs aux bornes, formats de données particuliers comme XML ou UTF-8, taille d'une structure). Cette grammaire du format des échanges peut-être étendue avec une description des états pour identifier les transitions possibles en fonction des sorties de la cible.

Le fuzzing n'est pas une technique très populaire dans la communauté scientifique car cette technique se construit sur la génération de tests aléatoires en un temps limité (*i.e.* il n'y a aucune preuve que l'absence de failles prouve la sécurité de la cible). Il existe à ce sujet peu de bases théoriques [10]. L'efficacité provient d'abord de l'expérience de l'auditeur à utiliser des politiques de modification qui soient efficaces pour la cible.

L'approche classique d'un fuzzing pour étudier le résultat d'une séquence invalide consiste à vérifier que la cible fonctionne encore correctement après chaque envoi (processus encore vivant, machine répondant à certaines commandes, etc.). Mais travaillant avec un protocole en boîte noire, l'instrumentation de la cible afin de détecter des faiblesses est difficile en l'absence d'accès aux instances. Donc seules les réponses (et l'absence de réponse) aux requêtes permettent de détecter une incohérence pouvant mener à une faiblesse.

Un de mes objectifs est aussi de vérifier la validité / conformité du protocole, c'est-à-dire si le résultat renvoyé est cohérent avec la spécification le décrivant. Afin de vérifier la conformité des réponses, nos travaux se sont appuyés sur l'ajout d'un oracle, dont le rôle est de générer la réponse attendue à partir de la requête, et ainsi permettre de

2. http://peachfuzzer.com/

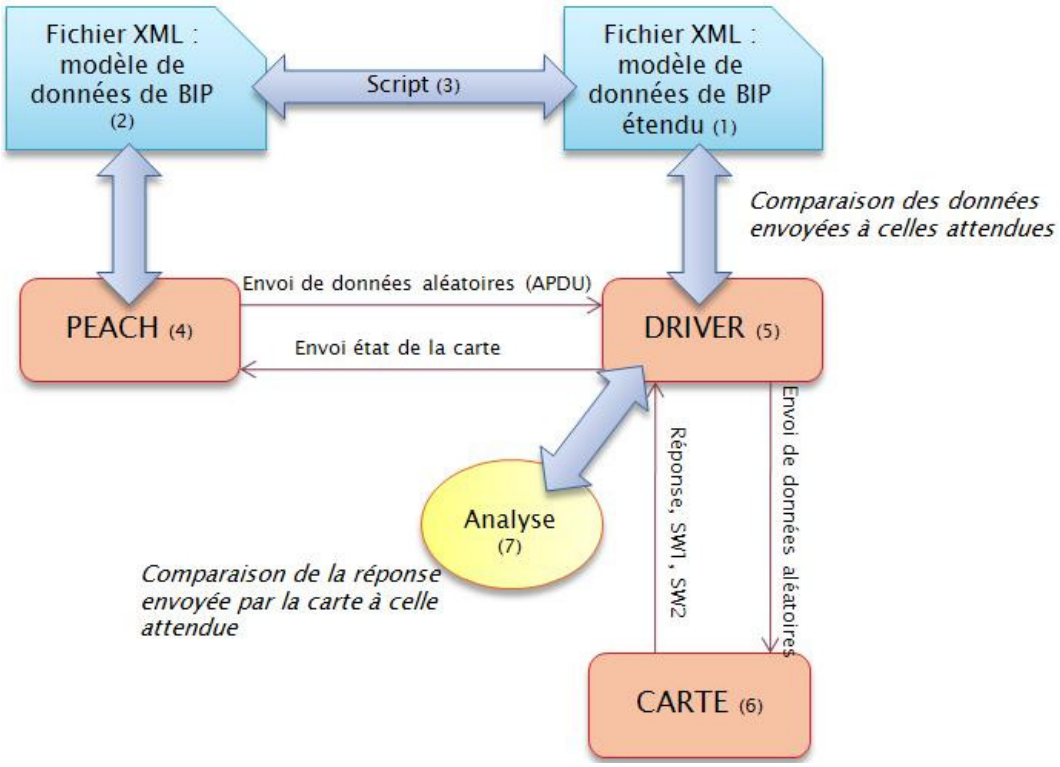


FIGURE 3.2 – Flot de fuzzing sur Java Card

détecter une différence vis-à-vis du comportement attendu. Il peut-être opposé que le développement d'un oracle ajoute un surcoût, qu'il aurait mieux fallu investir dans un *model checker*. Mais un oracle n'a pas besoin d'être une description complète des réponses et peut se limiter à un nombre réduit de valeurs contenues dans certains champs de la réponse.

La proposition d'architecture pour fuzzer en boîte noire les deux protocoles est donc une approche hybride combinant fuzzing et test (par l'ajout d'un oracle). Ma contribution scientifique est de montrer la viabilité de la méthode de fuzzing contre des protocoles de communication pour cartes à puce. Plus précisément, la contribution principale résultante de ces travaux est l'adaptation d'une approche fuzzing par génération de séquences malicieuses à partir d'une grammaire du protocole avec deux objectifs :

réduire l'espace de tests par adaptation en fonction de la cible ;

analyser les réponses par oracle, afin de valider la conformité au protocole étudié au lieu de se limiter à chercher une erreur spécifique (code de retour, absence de réponse).

L'architecture générale du banc de test des deux protocoles est présentée à la figure. 3.2. Peach génère à partir d'un fichier de grammaire les séquences malicieuses. Ces séquences sont envoyées par le pilote logiciel à la carte et les réponses sont retournées à l'oracle afin de vérifier si ce sont celles qui sont attendues. Comme Peach s'appuie un schéma XML pour décrire le format et les états du protocole, le schéma a été étendu pour associer à chaque état une description de la réponse attendue par l'oracle.

BIP

Le rôle du protocole BIP est de maintenir l'état de la connexion TCP, permettant d'éviter d'implémenter une pile TCP/IP dans la carte à puce. Le téléphone contient alors une passerelle BIP jouant le rôle de convertisseur de protocole. Il intercepte les commandes BIP envoyées par la carte et les convertit en commandes TCP afin de les renvoyer à l'application HTTP.

En mode serveur, la carte communique avec le téléphone à l'aide de commandes BIP. Ces commandes sont constituées de un ou plusieurs TLV (Tag Length Value) correspondant à un tag et une taille de données. Elles font partie de la technologie SIM Application Toolkit (SAT), un ensemble de commandes permettant de définir le comportement de la carte avec le monde extérieur.

Pour le protocole BIP, la construction du fichier de grammaire est construite de manière manuelle à partir d'un ensemble d'échanges valides et de la spécification BIP. La configuration de Peach pour fuzzer BIP définit les états du protocole et le format des paquets (les transitions).

HTTP

Le protocole HTTP est sans état et construit sur un format texte avec des requêtes volumineuses (comparées à des APDU), les nombreuses entêtes ont chacune un comportement particulier qu'il faut caractériser. De plus, il existe une difficulté majeure dans l'interprétation des réponses. En effet, les ressources contraintes et la volonté de limiter la surface d'attaque obligent les développeurs du protocole à oublier voire modifier des parties de la norme HTTP.

Il est donc nécessaire de connaître l'implémentation réelle du serveur web implémenté dans les cartes à puce pour produire un automate "taylorisé" pour la complexité de l'oracle HTTP et pour réduire la grammaire du protocole. Cette caractérisation possède certaines difficultés :

- Certains entêtes et valeurs associées doivent être fixés à partir d'une observation des échanges avec la carte ;
- Le protocole implémenté ne respecte pas forcément la norme HTTP : absence de certaines fonctionnalités, réponses négatives ou invalides à certaines requêtes, etc ;
- Certaines fonctionnalités de HTTP ne sont pas forcément présentes selon la terminologie utilisée dans la norme indiquant l'obligation ou non de leur présence (c'est-à-dire les mots « SHOULD », « MAY » et « MUST » de la RFC) ;
- Certains entêtes possèdent une certaine logique (la notion de cache par exemple) dont il faut identifier les caractéristiques.

Afin de caractériser la cible le plus précisément, l'outil PyHAT a été développé pour connaître les parties de la norme implémentée et le comportement réel. PyHat envoie une série de séquences afin d'identifier quel est le support de la norme HTTP réellement implémenté par le serveur :

- détection des méthodes HTTP implémentées (GET, POST, TRACE...) ;
- détection des versions de HTTP supportées ;
- détection des méthodes d'encodage de contenus supportés ;
- sensibilité à la casse des requêtes ;
- détections des entêtes analysées par le serveur.

La plus grande difficulté est de pouvoir interpréter les résultats retournés, en sachant que certaines réponses n'indiquent pas s'il y a erreur ou que le serveur web ne respecte

par obligatoirement les RFCs. Pour chaque méthode, version de HTTP et entête, il a fallu trouver un algorithme d'évaluation. Par exemple, l'outil soumet plusieurs requêtes avec des casses différentes afin d'identifier le support réel par le serveur. Certains cas sont d'ailleurs indécidables et le programme considère par défaut que la carte supporte ce cas.

Une fois cette analyse effectuée, PyHat génère une grammaire réduite et dont certains paramètres sont fixés. Cette grammaire est ensuite exploitée par Peach afin de générer des requêtes HTTP "fuzzées" qui seront envoyées à la carte. L'oracle se charge de vérifier les codes d'erreurs retournées et la cohérence des entêtes des réponses.

La détection des vulnérabilités se base sur l'analyse des codes de retour, à savoir : le code de retour BIP, le code de retour APDU *status word* contenu dans la réponse BIP et le code de la réponse HTTP en fonction des données envoyées en entrée.

Bilan

Il est intéressant de voir que l'idée d'utiliser une approche fuzzing contre les protocoles d'une carte à puce est relativement récente dans la communauté scientifique. Par exemple, le fuzzing du protocole EMV a été présenté simultanément à mes travaux [28] et Smart Brute, un outil de fuzzing des commandes APDU pour la recherche de codes non documentés a été récemment mis à disposition du public [3].

Le fuzzing du protocole EMV montre une approche sensiblement identique à la celle que je propose et démontre ainsi la validité de ce fuzzing "personnalisé". Dans le cas de BIP [CNC1, ST2], l'implémentation possède quelques différences de comportements vis-à-vis de la spécification. Pour le serveur HTTP embarqué [CN2, ST2], l'implémentation de la norme est incomplète sur certains points (comme la présence de réponse malgré l'absence du champs "Host") et des réponses sont non conformes (par exemple avec un code de retour HTTP 500 lors de la présence d'une entête "If-Match" invalide). De plus, des faiblesses dans l'autorisation associé aux commandes HTTP PUT et DELETE permettent de supprimer des ressources normalement non-accessibles par un utilisateur normal.

Plus généralement, les deux études ont aussi montré soit la présence de réponses avec un code APDU signalant une erreur interne (par exemple `0x6f00` pour signaler une exception Java Card), soit l'absence de réponse (signe d'une possible faiblesse en disponibilité) [4].

D'un point de vue critique, il existe certaines faiblesses dans les travaux exposés. Premièrement, Peach est utilisé avec les règles de mutations de champs par défaut (à quelques exceptions près) : test aux bornes pour les entiers, utilisation de séquences invalides, exploitation de transitions non valides, manipulation de champs indiquant une taille de structure, etc. Il aurait été intéressant de voir si d'autres politiques malicieuses étaient possibles. Mais comme évoqué plus haut, il existe peu de fondements théoriques sur le fuzzing, et un évaluateur se base principalement sur une intuition et des retours d'expériences [53]. Deuxièmement, la complexité (en terme de cas à gérer) des RFC HTTP rend difficile le développement d'un oracle capable d'identifier des cas intéressants dans l'ensemble des réponses retournées (de par la présence d'un nombre important de faux positifs). Troisièmement, l'approche de fuzzing à partir d'une grammaire est construite à partir des normes étudiées, mais pourrait aussi être appliquée sur la manipulation

3. https://www.thc.org/thc-smartbrute/

4. la remise à zéro de la carte ne prouve pas totalement la découverte d'une faille, car cela peut-être un échec dans l'exécution du code mais aussi un mécanisme de défense devenant actif

d'échanges enregistrés. Ainsi, il serait possible de tester des cas non documentés par la norme mais acceptés de facto par la carte et le logiciel l'interrogeant.

Néanmoins il n'y a pas eu de remise en cause de la confidentialité des secrets de la carte, l'inviolabilité de la Java Card n'étant pas remise en question. Cette résistance vient aussi des mécanismes de sécurité interne mais aussi des approches de développement. Par exemple, la présence de code d'exception Java Card dans les réponses APDU laissent supposer que les implémentations sont faites en Java Card, et donc bénéficie de la sécurité offerte par la JCVM.

3.2 Évaluation par applet frauduleux

Contrairement aux travaux sur le fuzzing, les hypothèses sur la capacité de l'attaquant sont plus importantes : l'attaquant possède le droit de charger un applet de son choix (c'est-à-dire que l'évaluateur possède les droits nécessaires, le plus souvent des clés GlobalPlatform) et la Java Card n'est pas équipé d'un vérificateur de *bytecode*. L'hypothèse d'un tel modèle d'attaquant est réaliste pour deux raisons.

Premièrement, le vérificateur de bytecode est défini comme étant un composant optionnel [5] jusqu'à la version 2.2 du standard Java Card. Dans la pratique, il peut être absent pour des raisons de coût car l'implémentation de l'algorithme demande une surface de silicium plus importante (certains industriels parlent du chiffre de 40 % de surface supplémentaire). Pour garantir l'innocuité des applications chargées, l'intégrateur les contrôle par un vérificateur de *bytecode* externe et garantit la chaîne de confiance par signature jusqu'au chargement dans la carte.

Deuxièmement, l'accès facile à des cartes développeurs permet à l'évaluateur d'acquérir pour des prix abordables des instances de cartes Java Card . Un développeur peut alors charger, exécuter et décharger des applets sans restrictions, le plus souvent à travers le protocole Global Platform (les clés de chargement étant en général connues et publiques). En l'absence de BCV, l'intérêt de l'attaquant est de pouvoir caractériser le comportement (fonctionnement et contre-mesures) d'une Java Card en chargeant des applets sous son contrôle.

Faiblesse de la spécification Java Card

Les travaux connus d'attaques logicielles contre des Java Card au moment de mes travaux (décrits en section 2.1) concernent des faiblesses d'implémentation des transactions et des interfaces partageables sur les cartes équipées de BCV. Ces attaques montrent que certains mécanismes sont plus difficiles à implémenter car la norme ne documente pas ces difficultés.

Les auteurs ont aussi exposé de possibles confusions de type en l'absence de BCV, par exemple avec une conversion de type (*byte as short array*), ou l'accès à un objet comme un tableau pour accéder aux entêtes des objets (*object as array*) et ainsi manipuler des références des membres de cet objet. Les deux impacts que les auteurs identifient sont la possibilité sur certaines Java Card (identifiées comme "d'ancienne génération" par les auteurs) de manipuler ses références (par remplacement ou génération de références,

5. Dans les premières versions des *drafts* de Java Card 3, la présence d'un vérificateur de *bytecode* est obligatoire dans la carte

c'est-à-dire *i.e.* de l'arithmétique sur les références). Mais leurs résultats sont difficilement identifiables ; Il est par exemple mentionné la possibilité de manipulation du *global AID registery* (une structure de données globale contenant la liste des AID des applets) mais sans confirmation de la réussite de cette opération. Pour sa part, la génération de références contiguës donne des informations incohérentes qui ne semblent pas représenter le contenu du plan mémoire. Les auteurs mentionnent alors la possibilité de protections internes de la carte comme :

— *array bound checking* et *object bound checking* pour vérifier que les accès à des membres d'un tableau ou d'un objet ne débordent pas de la taille de leur instance ;
— *integrity checking* pour vérifier avant chaque utilisation d'une référence si celle-ci n'a pas été modifiée.

Dans un premier temps, mes travaux ont montré d'autres confusions de type possible entre une référence et un entier sur la pile Java Card [CG1]. L'idée est de modifier le bytecode afin de mettre sur la pile JCVM une référence (instruction `ALOAD`) dans une position où elle sera retournée à la fin de la méthode comme un entier de type *short*. Une référence étant encodée sur la même taille qu'un short (16 bits), elle est directement utilisable avec ce type et peut donc être retournée dans un APDU. En présence d'un BCV, un tel code serait refusé car un vérificateur de *bytecode* est capable de connaître le type des éléments attendus de la pile JCVM (et donc le type de l'élément retournée) pour le comparer avec celui du type attendu par la signature de la méthode.

De nos travaux et ceux de Poll, la possibilité de confusion de type signifie qu'il est possible lire et écrire les valeurs internes de certaines références. Cette faiblesse est normalement impossible en présence d'un BCV car il n'existe aucune fonction native permettant d'accéder à la valeur interne d'une référence (comme une hypothétique méthode `short refToShort(Object o)`).

Outrepasser le pare-feu par les instructions statiques

Les exemples de confusion de type présentées dans la précédente section ont un intérêt limité car le pare-feu de la JVCM empêche d'accéder en dehors de son instance d'applet. L'impact se limite donc à la connaissance des méta-données (les entêtes de tableau et d'objets) associées aux instances de l'applet malicieux.

La norme Java Card impose la présence d'un pare-feu dans les cartes à puce Java Card. Le JCRE doit vérifier dynamiquement (pendant l'exécution) que les références manipulées par l'applet appartiennent à son contexte d'exécution. En terme d'implémentation, ce mécanisme d'isolation a un surcoût limité en temps d'exécution, car une partie des contrôles sont déjà effectués par le BCV. Par exemple, l'édition des liens lors du chargement de l'applet dans la carte vérifie que chaque opérande de *bytecode* avec un paramètre représentant un index dans le *Constant Pool Component* pointe sur un élément existant avec un type attendu par l'opérande.

Il existe une seule exception à l'isolation par le pare-feu : les instructions Java Card manipulant des références vers des membres statiques de classe (spécifiquement `getStatic`, `putStatic` et `invokeStatic`). Elles ne sont pas soumises au pare-feu et il n'y a pas de changement de contexte par le JCRE lorsque l'une de ses instructions accède à un champ statique d'un autre contexte. Cette exception au principe d'isolation du pare-feu provient du fait que certains objets statiques du système doivent être accessibles depuis tous les contextes, par exemple le tampon d'entrée sortie `APDU`.

Dans l'hypothèse d'une absence de BCV, les attaques par confusion de types permettent d'avoir la valeur interne des références de membres statiques de classes d'autres applets ou du système (comme la méthode statique `getCurrentAPDU` de la classe `APDU` retournant le tampon d'entrée/sortie partagé entre les applets). En utilisant ces valeurs comme paramètres des méthodes manipulant des références vers des membres statiques, il est donc théoriquement possible d'accéder à n'importe quelle référence existante dans la carte.

Le format CAP est un format compact, ne contenant que des identifiants numériques à la place des noms complets de classe et de *package*. Ces identifiants sont disponibles dans les fichiers EXP (pour *EXPort*), afin d'indiquer au convertisseur quel est l'identifiant numérique pour chaque classe et membre de classe de l'interface public offerte par le fichier CAP associé. Ainsi, le kit de développement contient plusieurs fichiers EXP indiquant les identifiants numériques de chaque élément public de l'API Java Card. Durant le chargement du fichier CAP dans la carte, une édition des liens est réalisée afin de remplacer ces identifiants numériques par les références associées.

Dans le cas des instructions agissant sur des membres statiques, elles possèdent un paramètre représentant un index pointant dans le *Constant Pool Component* afin de connaître les références d'une part la classe et d'autre part la variable (dans le cas de `getStatic` et `putStatic`) ou la méthode (dans le cas de `invokeStatic`). Durant l'édition des liens, le format CAP contient un *Reference Location Component* contenant l'adresse des instructions qui possèdent comme paramètre un index pointant sur une entrée du *Constant Pool Component*. L'édition des liens parcourt donc ce *Reference Location Component* et remplace l'index du paramètre de l'instruction par la référence associée.

> La vulnérabilité découverte concerne l'opération d'édition des liens. Sur certaines implémentations de Java Card, ce mécanisme peut avoir une confiance trop importante sur les valeurs contenues dans le *Reference Location Component*. Dans ce cas, si une entrée de ce composant est retirée, alors le paramètre pointé par cette entrée ne sera pas mis-à-jour. Ainsi un attaquant peut charger un fichier CAP avec une instruction manipulant un membre statique dont la référence passée en paramètre est sous le contrôle de l'attaquant.

Impacts

Je présente maintenant les différentes attaques EMAN qui ont découlées de cette faiblesse [CH1].

Attaques EMAN

Mon travail a porté dans un premier temps sur **EMAN1** [JIC2, CG1, CG2], la première application possible de cette faiblesse. Le principe est de générer une suite d'applets contenant une fonction retournant la valeur empilée sur la pile par la méthode `getStatic` avec comme opérande une valeur qui sera considérée comme une référence interne par la carte. Mais comme ce processus est long (chargement / déchargement d'applet), il a été optimisé par l'usage de la confusion de type afin de pouvoir exécuter une méthode statique contenue dans un tableau sous le contrôle de l'évaluateur.

La méthode `putStatic` permet aussi de modifier des emplacements mémoires identifiées lors de l'étape précédente, et ainsi de modifier le code d'un autre applet comme par

exemple remplacer des instructions par des NOP, voire de les remplacer par appel vers du code contenu dans notre applet.

Dans un second temps, mes travaux ont mené à l'attaque **EMAN2** [CIC3] qui permet de modifier la pile Java Card des applets en cours d'exécution afin de pouvoir exécuter du code malveillant contenu dans un tableau. Une fois l'adresse de la pile identifiée par l'attaque EMAN1, il est possible de remplacer avec la méthode de EMAN1 l'adresse de retour contenue dans une *frame* de la pile par l'adresse du code malveillant, à la manière d'un *stack overflow* [107]. Récemment, Faugeron a détaillé une attaque similaire par *buffer underflow* [6] et Bouffard *et al.* montre qu'une méthode de protection par typage des éléments de la pile Java Card peut être outrepassé dans les mêmes conditions [3].

Mais un appel à une méthode de l'API Java Card doit utiliser une référence interne, normalement inconnue car insérée dans l'applet lors de l'édition des liens dans la carte. Un autre travail associé à notre équipe [19] propose une méthode pour obtenir la valeur des références internes des méthodes de l'API. Par défaut le *Reference Location Component* d'un fichier CAP contient les offsets qui doivent être remplacés par des références internes. Normalement ces offsets sont des opérandes représentant une entrée dans le constant pool, et sont donc manipulés par des instructions nécessitant une résolution d'une entrée du *Constant Pool*. L'attaque consiste à remplacer l'instruction par une autre manipulant une opérande comme un entier (par exemple `sspush`), afin de pousser sur la pile la valeur des références internes pointant sur les fonction proposée par l'API dans la carte. Ainsi, il est possible de développer du *bytecode* complexe capable d'interagir avec le système (au lieu de se limiter à la lecture / écriture en mémoire), à la manière des *shellcodes*. L'étudiant Tiana Razafindralambo a développé dans notre équipe une extension de ce travail en mutant un code structurellement correct en un applet par corruption de l'éditeur de liens Java Card embarqué [WN1, 21].

Les attaques EMAN1 et EMAN2 permettent donc de lire et écrire à des adresses correspondantes aux références internes forgées par l'attaquant et à exécuter du *bytecode* contenu dans un tableau, donc ne respectant potentiellement pas le format du *bytecode* attendu par la JCVM. Néanmoins, l'ensemble de ces opérations s'effectuent au sein de la JCVM. L'attaque EMAN3 proposée par Guillaume Bouffard *et al.* [1] propose de dépasser cette limitation en exécutant du code natif embarqué dans un applet en le faisant passer pour une méthode native de la carte. En effet, au sein de la carte se trouve un ensemble de méthodes appelables par des applets mais qui exécutent du code natif, à la manière de JNI dans le cas de java. L'attaque est d'identifier dans les plans mémoires extrait de la carte par l'attaque EMAN1 la table d'indirection des méthodes natives. Une fois identifiée, une entête de méthode native et une entrée dans la table d'indirection sont ajoutées pour l'accès à cette méthode. L'adresse d'exécution du code native est alors une structure sous le contrôle de l'attaquant, en général dans un tableau chargée dans l'applet.

Une autre possibilité d'exploitation réalisée par Guillaume Bouffard *et al.* [12] prend l'hypothèse d'un attaquant interne capable de modifier la configuration du convertisseur. Elle n'exploite pas la faiblesse des fonctions manipulant des membres statiques mais exploite un comportement spécifique du convertisseur (utilisé *off-card*). Plus précisément, durant la traduction des fichiers class de l'application en un fichier CAP, le convertisseur résout pour chaque entrée du *Constant Pool* le symbole associé dans les fichiers EXP. Le problème provient du fait que le convertisseur prend la première référence trouvée, donc un attaquant peut ajouter dans le fichier EXP d'une librairie une entrée sur une référence

de l'API native embarqués dans la carte. Ainsi, même si l'applet est considéré comme valide d'un point de vue du BCV et du développeur de l'applet, cette conversion biaisée permet à l'attaquant d'introduire son code dans la carte.

La notion d'applet mutant

Comme il est possible d'extraire l'organisation mémoire (*i.e.* avoir un plan partiel ou complet de la mémoire), il est possible d'étudier les possibilités d'attaques en fautes sur les cartes (décrites en section 4.2), car les conséquences de l'impact d'une faute sur le contenu mémoire sont identifiables. Notre équipe a proposé la notion d'applets mutants comme l'exploitation d'applets valides (du point de vue du BCV), mais devenant malicieux après une attaque en faute. L'attaquant, s'il contrôle la création de l'applet, peut alors ajouter énormément de logique de son attaque dans l'applet et utiliser une seule attaque en faute pour déclencher cette charge logique. Un attaquant peut aussi créer un programme avec de larges portions mémoires contenant des symboles particuliers puis en extraire le contenu après une faute afin d'identifier la position de son applet grâce aux impacts des modifications engendrées par l'attaque en faute.

Un exemple simple d'attaque physique sur du bytecode Java Card est de remplacer des portions de code de tests de sécurité par des zéros, équivalente à des instructions NOP (sans effet). Mes travaux ont abouti au moyen de combiner nos attaques logiques EMAN et une attaque en faute avec **EMAN4** [CIC3]. Ce travail part de l'observation que certaines implémentations de Java Card conservent les tableaux d'octets statiques à la suite des méthodes d'instructions par classe. Si une de ces méthodes contient une boucle, celle-ci finira par une instruction `goto_w` suivie de la taille du saut généralement négatif (par exemple `0xFF17`). Si une faute est réussie sur le premier octet de l'opérande de l'instruction de saut (par exemple `0x0017`), alors le saut est maintenant positif et peut sauter dans le tableau statique placé après la méthode. L'attaquant peut alors faire exécuter n'importe quel bytecode invalide (d'un point de vue BCV) car vu comme des constantes d'un tableau

Après avoir présenté cette notion d' "applet mutant" à la communauté scientifique, de nombreux chercheurs ont repris ce travail pour montrer les possibilités d'une telle combinaison. Notre équipe a proposé des outils pour générer des applets malicieux selon un modèle de faute précis [21]. Barbu *et al.* [34] ont montré la possibilité de perturbation de l'instruction `checkcast` qui permet d'accéder à la représentation native d'un objet. Les mêmes auteurs proposent dans [24] plusieurs attaques par applets mutants en perturbant physiquement une valeur lors de son empilement sur la pile d'opérande. Dans le cas de Java Card 3, Barbu *et al.* [25] ont aussi proposé d'attaquer le *multithreading* en noyant une application sous un flot d'E/S afin de forcer celle-ci à s'arrêter à un instant particulier, ou encore en forgeant avec une faute une référence pointant dans la *frame* de l'application ciblée.

3.3 Bilan

Mes travaux avec l'équipe SSD ont permis d'identifier l'existence d'une vulnérabilité dans la spécification Java Card, permettant d'outrepasser l'isolation offerte par le JCRE entre applets ou vis-à-vis du système hôte. Nous avons démontré l'existence de cette faiblesse sur plusieurs instances de Java Card de plusieurs constructeurs différents si l'attaquant maîtrise le contenu. Plusieurs méthodes d'exploitation de cette vulnérabilité

permettent de lire voire modifier le contenu d'autres applets (EMAN1), d'exécuter du *bytecode* contenu dans un tableau (EMAN2) voire d'exécuter des méthodes natives sous notre contrôle (EMAN4). En 2014, la présence de cette faiblesse a encore été confirmée par Guillaume Bouffard sur plusieurs cartes [1], même si la classe d'attaques EMAN fait aujourd'hui partie du processus de certification des Java Card.

Il est intéressant que mes travaux ne ont pas les seuls à avoir montré la possibilité d'exécuter un applet sous le contrôle de l'attaquant. Nohl [14] a démontré en 2013 la possibilité de pouvoir charger un applet en abusant du système OTA (*over-the-air*, un mécanisme de mise à jour des applications par SMS). Certaines implémentations utilisent une signature des messages SMS avec une cryptographie faible, permettant à l'auteur de charger un applet malicieux et de prendre contrôle de la SIM (l'auteur parle de 13% du parc de cartes SIM déployés aux états-unis qui seraient exploitables). Il est malheureusement très difficile d'identifier les faiblesses exploitées sur les cartes pour outrepasser les défenses de la machine virtuelle car elles ne sont pas clairement exposées publiquement, l'auteur évoquant des faiblesses dans l'implémentation de la machine virtuelle. Il est donc difficile de voir si mes travaux ont un rapport ou si d'autres vulnérabilités existent dans les cartes.

La criticabilité de la vulnérabilité levée par mes travaux est néanmoins à nuancer selon plusieurs points. Premièrement un applet frauduleux ne doit pas être vérifié par un BCV avant d'aboutir dans la carte. Des cartes équipées de BCV ou un chargement authentifié de l'applet après validation par BCV externe permettent de répondre à cette menace. Deuxièmement, les expérimentations ont montré des mécanismes de sécurité implémentés dans la carte, la carte refusant de charger l'applet ou devenant muette de manière temporaire ou permanente (self-destruction). Il existe donc probablement des contre-mesures limitées pour détecter des menaces bien précises au chargement sans nécessairement implémenter un vérificateur de *bytecode* complet. Dans notre cas, la contre-mesure la plus évidente est de vérifier la cohérence du *Reference Location Component* vis-à-vis de l'ensemble du *bytecode* pour éviter l'attaque sur l'opérande des instructions de membres statiques (nous parlons alors de BCV partiel).

L'impact de cette faiblesse réside dans la disponibilité de cartes développeurs pouvant mener à la découverte des mécanismes internes de la carte, c'est-à-dire les plans mémoire de la ROM (contenant le système d'exploitation, la JCVM, le JCRE et les applications livrées avec une carte vierge), la RAM (contenant les états du système et la pile d'exécution Java Card lorsque la carte est connectée à un lecteur) et/ou l'EEPROM (contenant les applications installées après émission de la carte). L'analyse de ces plans mémoires permet d'ouvrir la porte à la rétro-conception de programmes afin d'identifier d'autres faiblesses [4, 2]. Comme les instances de Java Card en circulation sont des versions plus récentes, il est possible d'envisager la recherche de vulnérabilités à partir des binaires extraits.

3.4 Ressources

Outils

Accès à la carte

OPAL [6] est l'une des premières implémentations de la norme Global Platform [87] Ce protocole n'était exploitable que par des outils propriétaires et il était impossible de pouvoir automatiser les attaques EMAN avec ceux-ci. J'ai mené le développement et mis en place un suivi qualité (tests unitaires, métriques sur le code) de cet outil. Avec l'aide de plusieurs étudiants, nous avons implémenté les commandes les plus utilisées, avec l'ensemble des modes disponibles dans la norme : SCP01, SCP02 et SCP03. OPAL possède également la seule implémentation open-source de la norme RAM-over-HTTP à ce jour [CI3].

Cet outil est fonctionnel et testé avec plus d'une vingtaine de cartes. Il est aussi capable de supporter des variantes propriétaires de la norme. Il est utilisé par de nombreux industriels, ce qui montre que cet outil est aujourd'hui reconnu et adapté aux besoins de nombreuses entreprises du secteur de la carte à puce.

Cap Malicieux

Une grande partie de nos travaux concerne la création d'applets explicitement malicieux (lorsque la carte n'est pas équipée par d'un BCV) ou d'applets mutants. La génération manuelle de fichiers CAP possédant ces propriétés étant laborieuse, j'ai mené le développement du logiciel CapMap [7]. Une première version de ce logiciel est capable de désérialiser un fichier CAP en une suite d'objets manipulables directement par un programmeur. Ce dernier peut alors manipuler le contenu du fichier et sérialiser une nouvelle version dans un fichier CAP. Cet outil a été conçu dans la même philosophie que les bibliothèques d'introspection de fichiers "class" comme BCEL, en étant toutefois capable de travailler directement sur des fichiers Java Card. Cet outil est disponible en open-source et est déjà utilisé publiquement par des équipes de recherche.

Une extension de cette bibliothèque (uniquement disponible pour certains partenaires) permet la mise en cohérence des dépendances internes d'un fichier CAP après une modification. En effet, l'ajout d'instructions dans le bytecode ou la modification de paramètres des *Components* (par exemple la liste des offsets des références statiques exploitées dans les attaques EMAN) nécessite de recalculer un grand nombre de champs du fichier CAP (taille des composants, actualisation des instructions référençant les entrées du constant pool et réorganisation des index, etc.). Cette extension apporte donc une très grande valeur ajoutée à la bibliothèque CapMap dans le cadre d'évaluation d'attaques par application malicieuse car ce sont des opérations qui nécessitent une grande connaissance de la sémantique du fichier CAP.

Participants

Dans le cadre de mon co-encadrement de la la thèse de Nassima Kamel [ST2], les outils et travaux de fuzzing BIP [CNC1] ont été développés par Matthieu Barreaud, ingénieur. Le fuzzing du protocole HTTP [CN2] a été réalisé par les étudiants de M1

6. https://bitbucket.org/ssd/opal/
7. https://bitbucket.org/ssd/capmap-free

Amine Belhociné, Jérémie Clément, Nicolas Tarriol, Romain Séverin, Mamadou Lamine Balde et Lylia Tikobaini. Tous ces outils ont été transférés à notre partenaire Gemalto.

Dans ce chapitre, je présente une faiblesse de la spécification Java Card pouvant mener à la possibilité d'accéder en lecture et écriture à d'autres applets dans le cas où la carte n'est pas équipée de BCV [JIC2, CG1, CG2]. J'ai encadré le développement des attaques EMAN en grande partie durant des projets de Licence 3 Informatique, Master 1 Informatique et Master 2 Cryptis de l'Université de Limoges avec le professeur Jean-Louis Lanet. L'attaque originale EMAN1 fut réalisée par Émilie Faugeron (EM-) avec Anthony Dessiatnikoff (-AN), ce dernier ayant aussi contribué à un outil d'analyse du contenu du dump mémoire. Par la suite, les travaux autour d'EMAN ont principalement été développés par Guillaume Bouffard [CIC3, CH1], avec les apports de Tiana Razafindralambo [WN1, 21].

Ces travaux ont nécessité le développement de deux outils développés sous ma direction. OPAL a été développée par Eric Linke et Damien Arcuset pour le support de SCP01 et SCP02, et Anis Bkakria pour RAM-over-HTTP [CI3]. Diverses améliorations et IHM d'OPAL ont été proposés par Julie Rispal, David Pequegnot, Guillaume Bouffard, Valentin Cassair, Christophe Bouygues, Moustakim Mehdi et Maxime Chazalviel. CapMap a été développé par Guillaume Bouffard et Julien Boutet.

Chapitre 4

De la sécurité de la machine virtuelle Java Card

"You – shall not – pass !"

— Gandalf, *Lord of the Rings*

4.1 Introduction

Après avoir présenté différentes menaces logicielles dans l'axe 1 (voir le chapitre 3), ce chapitre détaille un ensemble de mécanismes à destination des mécanismes internes d'une machine virtuelle Java Card.

Dans une première partie des travaux décrits en section 4.2, je propose plusieurs méthodes de protection contre les attaques en fautes des applications par l'ajout de vérifications lors de l'exécution de la machine virtuelle.

Dans un second temps, nous nous intéressons au problème de mise à jour dynamique des applications. La section 4.4 détaille une extension de l'architecture Java Card afin de mettre à jour les applets chargés dans la carte tout en conservant les instances en cours d'exécution.

4.2 Contrer les attaques en faute contre les applets

Il existe un historique important en terme de publications scientifiques contre les fautes sur le matériel dans le domaine spatial et aéronautique afin de garantir la sûreté de fonctionnement. L'influence des rayonnements cosmiques sur les composants électroniques [112] nécessite le développement de contre-mesures permettant de détecter et/ou corriger le risque qu'un ou plusieurs bits en mémoire puissent changer de valeur de manière aléatoire (on parle alors de *single event upset* ou *SEU*).

Il existe différentes protections logicielles et matérielles contre les SEU afin de garantir la sûreté de fonctionnement [29]. Elles sont classifiées selon :

- le type de redondance : *space-redundant* (duplication du matériel) ou *time-redundant* (duplication de l'exécution au niveau logiciel) ;
- la résistance à un type de faute : *hard errors*, *intermittent errors* ou *transient errors* ;
- la hiérarchie dans le système : au niveau électronique, du système d'exploitation ou de l'application (il existe aussi des solutions hybrides).

Au niveau des mémoires, la principale approche est l'utilisation de la détection et correction d'erreur (*Error Checking and Correcting* ou *ECC*). Dans le cadre de mes recherches sur la sécurité de la virtualisation (voir section 6.4), le processeur étudié est

un LEON4 : un composant renforcé conçu pour le domaine spatial [32]. Il inclut des mécanismes pour détecter et corriger des erreurs aléatoires dans la RAM, les registres du processeur et dans les caches. Ces contre-mesures contre les fautes aléatoires détectent et corrigent jusqu'à 4 changements aléatoires de bits par mots de 32 bits sans surcoût en terme de performance.

Les contre-mesures exposées dans le domaine spatial furent réutilisées dans le domaine de la carte à puce contre les attaques en fautes. On peut citer plusieurs mécanismes de redondance au niveau matériel (*Simple Duplication with Comparison*, *Multiple Duplication with Comparison*, *Simple Duplication with Comple- mentary Redundancy*...) ou au niveau logiciel (*Simple Time Redundancy with Comparison*). Toutes ces méthodes sont de la responsabilité de l'encarteur et sont en général relativement peu documentées, une seule publication confirme la présence de ces contre-mesures dans les cartes actuelles [62].

Lors de la conception de contre-mesures, je considère que plusieurs objectifs doivent être suivis (déjà exposés dans le chapitre 2) :

— une contre-mesure doit être automatique : un programmeur n'a pas de contrainte dans sa manière de programmer, le support des mécanismes de défense doit donc être transparent et indépendant du code.
— une contre-mesure doit être peu coûteuse pour la carte à puce : son support nécessite une place minimale en terme de mémoire (EEPROM, RAM et ROM) et un surcoût minimal pendant l'exécution qui ne doivent pas dépasser 5 % comparés à une exécution sans contre-mesures.
— une contre-mesure doit être compatible avec le flot de développement d'un applet Java Card : outre la nécessité que la mise en place des contre-mesures doit être transparente pour le programmeur, il ne doit pas avoir d'impacts fondamentaux sur la structure du CAP file ou des API utilisées par le programmeur.

La première idée possible est de réutiliser les mécanismes logiciels et matériels de redondance probablement disponibles dans une carte pendant l'exécution des applets. Mais le surcoût en terme mémoire (doublement de la RAM et de l'EEPROM) et d'exécution nécessaire pour dupliquer l'exécution d'un applet ne répond pas à nos contraintes.

Une autre solution serait d'ajouter des mécanismes de contrôle pour le développeur d'applet. Les travaux de protection de code binaire dans une carte à puce par Akkar *et al.* [81] proposent l'ajout d'annotation de la part du développeur afin d'identifier des chemins d'exécution critique pour les vérifier pendant l'exécution dans la carte

Le développeur dispose plusieurs points de repères : le départ du chemin d'exécution, plusieurs points de passage et l'emplacement de la vérification du chemin exécuté. Avant la compilation, une phase de pré-traitement déduit l'ensemble des chemins d'exécutions possibles et les ajoute dans le binaire du programme, afin de pouvoir les vérifier pendant l'exécution. Cette approche a le défaut de nécessiter une compréhension des attaques en faute et de l'impact de telles fautes sur le matériel, la machine virtuelle et le code applicatif.

Mais l'ajout de ce processus supplémentaire, orthogonal au développement, nécessite aussi que le développeur soit capable d'identifier les chemins critiques et que la mise en place des différents points de repère soit efficace, nécessitant un surcoût en terme de formation et d'utilisation pour les développeurs. De plus, l'impossibilité de connaître l'impact réel d'une faute sur du code binaire et la présence (ou non) de contre-mesures sur la carte ne permet pas d'identifier l'efficacité obtenue.

Néanmoins cette idée est le point de départ de mes travaux sur la nécessité de construire une information sur la structure du code et de la vérifier pendant l'exécu-

tion. Afin de réduire le travail pour le développeur, mon approche est de travailler sur le code compilé (*bytecode*) et de construire les contre-mesures sans l'aide du développeur. Cette position se justifie aussi par le fait qu'une attaque en faute impacte la représentation binaire du code qu'il est difficile de voir par le développeur.

Dans la suite de ce document, je présente les quatre contre-mesures que nous avons proposées. Toutes ces contre-mesures possèdent des points communs :

— elles fonctionnent en générant *offline* des informations sur la structure du bytecode qui sont exploitées durant l'exécution de la machine virtuelle. Cette idée permet de déporter la grande majorité du coût de calcul de la contre-mesure en dehors de la carte.

— elles ne nécessitent aucune modification du bytecode (ou des modifications transparentes pour l'utilisateur car effectuées avant chargement ou dans la carte). Pour conserver les informations nécessaires pour les contre-mesures, un composant additionnel de taille réduite est généré après la création de l'applet et ajouté au fichier CAP comme *Custom Component*. Si la machine virtuelle ne supporte pas le mécanisme de protection, la norme précise que le *Custom Component* n'est pas exploité par la carte.

Modèle de fautes sur les cartes

L'ensemble de nos travaux de contre-mesures contre les attaques physiques sur le *bytecode* s'appuie sur un modèle de faute. Il doit être considéré comme réaliste mais nous ne possédions pas l'expérience ni les moyens d'expérimenter en pratique. J'ai d'abord chercher à modéliser les fautes et identifier celles pertinentes pour l'industrie. La thèse de Martin Otto [80] définit plusieurs paramètres pour caractériser une faute : la localisation spatiale de la faute (quel partie du processeur est affectée ?), la localisation temporelle de la faute (quelle est la précision temporelle de l'attaquant ?), le nombre de bits affectés (si applicable, quel est l'impact sur le contenu des registres et/ou en mémoire ?), le type de faute (transitoire ou permanent), le taux de succès et la durée de la faute.

Le cas le plus favorable à l'attaquant mais réaliste d'un point de vue technique à partir de la dichotomie d'attaques ci-dessous peut-être posé comme suit :

— une attaque en faute peut modifier les valeurs contenues dans une portion de la mémoire ou la valeur circulant sur un bus. Nous avons pris l'hypothèse qu'il était possible de modifier au mieux un octet. Il semble être difficile de pouvoir changer un seul bit car les décharges d'énergie d'un laser impactent l'ensemble d'un mot mémoire ;

— un attaquant peut choisir ce mot dans n'importe quelle position mémoire, que ce soit dans l'EEPROM, la RAM ou la ROM (dans ce dernier cas, il modifie la valeur lors de son transit entre la ROM et le processeur) ;

— la valeur injectée peut-être `0x00` ou `0xFF` selon le type de technologie de la mémoire. Mais la valeur injectée peut aussi être une valeur aléatoire car la mémoire de la carte peut-être chiffrée et il est alors impossible de prédire la valeur injectée car elle représente le chiffré de `0x00` ou `0xFF` ;

— un attaquant peut choisir un moment précis où envoyer son attaque. C'est le cas le plus favorable ;

— en revanche l'attaquant ne peut procéder à deux attaques en même temps. La difficulté d'obtenir suffisamment d'informations sur l'impact de deux attaques syn-

chronisées semble encore importante. À noter que cette situation peut évoluer dans un futur lointain avec l'amélioration des techniques d'attaques en faute.

On notera quelques limitations de ce modèle. Premièrement, il ne considère pas le cas de plusieurs fautes, soit par l'usage d'une faute unique mais répétés pendant la durée d'exécution, soit par l'application de plusieurs fautes pendant la durée d'exécution. À notre connaissance, il n'existe de cas concrets publics aujourd'hui sur l'exploitation d'une telle approche. Deuxièmement, il caractérise une faute en mémoire, et non sur d'autres éléments dans le microprocesseur (sur les bus par exemple). Mais je considère que l'impact est équivalent : un changement dans le résultat mémorisé en mémoire ou dans un registre. L'autre faute possible est le fait de bloquer le processeur pour passer une étape dans le code d'exécution. Du point de vue de code, cela signifie une attaque remplaçant une série d'instructions par du code sans effet (NOP). Je considère donc qu'un modèle de faute en mémoire suffit pour représenter différentes classes d'effets sur le microprocesseur.

Ainsi, le modèle d'attaquant considéré est considéré comme très puissant par Otto [80] à un point près : il ne possède pas la capacité de changer moins d'un octet, soit une réduction du modèle de *fault byte model*. Le fait que l'attaquant affecte au moins 8 bits est dû à la présence du chiffrement de la mémoire mais aussi au fait que la taille d'un laser pour une attaque en faute ne peut être inférieure à l'implémentation d'un mot en mémoire physique (de l'ordre de quelques μm^2).

Propositions

Une fois le modèle posé, mon travail en collaboration avec la thèse de doctorat d'Ahmadou Séré [ST4] a été de proposer des contre-mesures et de pouvoir quantifier leur métriques [JIC1, CIC6, WI4, CI6, CN5].

Remplacement des bytecodes sensibles

Le modèle de faute considère comme valeurs injectées `0x00` et `0xFF`. Dans une carte sans chiffrement de la mémoire, ces deux opcodes représentent respectivement l'instruction NOP et une instruction qui n'existe pas. Dans le cas d'une instruction qui n'existe pas (0xFF), la machine virtuelle peut détecter automatiquement cette erreur. Mais le cas du NOP est plus dangereux, car l'injection de cette valeur peut outrepasser des tests ou annihiler des portions de code. La première proposition est donc de remplacer cette instruction par un bytecode qui n'est pas attribué comme signalé par Bizzotto et Grimaud [92].

Champ de bits

La seconde proposition s'appuie sur l'idée qu'une attaque peut modifier des *bytecodes* de telle manière qu'une opérande d'une instruction Java Card peut devenir une instruction. Nous cherchons à détecter cette modification en vérifiant que nous n'exécutons pas une opérande mais bien une instruction.

La contre-mesure consiste à calculer off-card un tableau de bits représentant le type de chaque octet du bytecode : la valeur fausse indique que l'octet pointé par le PC est une instruction et la valeur vraie une opérande. Durant l'exécution, la machine virtuelle conserve la position dans le tableau de bits qui correspond à l'avancée dans le flot d'instructions. Avant chaque exécution d'une instruction, la machine virtuelle vérifie si le bit

associé est bien égal à la valeur 0. Si oui, alors elle exécute l'instruction sinon elle arrête son exécution car elle tente d'exécuter une opérande.

Basic block

La troisième proposition exploite les informations de sauts présents dans le code. Si l'attaque en faute modifie le graphe attendu d'appel de méthodes (Control Flow Graph ou CFG), alors il est possible d'outrepasser des portions de code critique (à la manière de l'injection de *bytecode* NOP).

Cette proposition est de vérifier qu'une suite d'instructions qui ne possède pas d'instruction de rupture de flots (saut, exception, retour d'une méthode) est bien exécutée sans altération (c'est-à-dire qu'une ou plusieurs instructions n'ont pas été modifiées et que la suite est complètement exécutée).

Le bytecode est décomposé en plusieurs blocs d'instructions. Chacun ne possède qu'un seul point d'entrée et qu'un seul point de sortie. Cette notion est connue dans le domaine des CFG sous le nom de *basic blocks*. L'algorithme de décomposition d'une suite d'instructions en *basic blocks* s'appuie sur la recherche de l'ensemble des leaders, c'est-à-dire l'ensemble des instructions qui peuvent être un point d'entrée (arrivée d'un saut, début d'une méthode, début d'un bloc de traitement d'une exception,...) ou un point de sortie (départ d'un saut, fin d'une méthode, fin d'un bloc de code, ...). Un *basic block* est une suite de *bytecode* commençant et finissant par un leader et ne contenant aucun leader.

Une fois l'ensemble des *basic blocks* construit, une somme de contrôle correspondant au XOR de chaque instruction est calculée pour chaque *basic block* et associé à la position du *basic block* (l'index de l'instruction de début du *basic block* et de fin du basic block).

Durant l'exécution de l'applet dans la carte, la machine virtuelle associe la position du pointeur d'instruction au *basic block* correspondant et à sa somme de contrôle. Lorsque la machine virtuelle entre dans un *basic block*, elle calcule le XOR de chaque instruction exécutée tant que la fin du *basic block* n'est pas atteint. Si à la fin de l'exécution du *basic block*, la somme de contrôle calculée n'est pas la même que celle mémorisée (ou si l'exécution est détournée avant la fin du *basic block*) alors soit une (ou plusieurs) des instructions exécutées ne fait pas partie du *basic block* original, soit seule une partie du code associé au *basic block* a été exécutée, soit une instruction de saut est intervenue avant la fin du *basic block* associé, soit des opérandes ont été modifiées.

Path checking

La quatrième proposition est une autre méthode pour lutter contre contre des sauts illégaux. Elle consiste à vérifier les chemins d'exécution possible du flot de programme avec un minimum de ressource mémoire.

Pour mémoriser un chemin à bas coût, la quatrième proposition est de calculer et de mémoriser uniquement les transitions possibles entre *basic blocks*. En dehors de la carte, l'ensemble des *basic blocks* sont calculés en utilisant l'algorithme décrit dans la troisième proposition. Ces chemins sont ensuite encodés sous forme binaire : De manière simplifiée, si le chemin correspond à un saut le chemin est encodé avec "1", et si le chemin correspond à la suite des instructions (c'est-à-dire qu'il n'y a pas de saut) le chemin est encodé avec "0". Les opérations de type `switch` sont encodées avec plusieurs bits.

Pendant l'exécution, la JCVM effectue la même opération : elle construit un tableau de bits en fonction des chemins pris par le programme, et plus exactement ajoute pour chaque instruction de saut conditionnel un "1" si le saut est effectué et un "0" dans le cas

Protection	Vitesse d'exécution	EEPROM	ROM
Champs de bits	$\approx +3\%$	$\approx +3\%$	+1%
basic block	$\approx +5\%$	variable (0 à 5%)	+1%
Patch checking	$\approx +8\%$	variable (0 à 10%)	+1%

FIGURE 4.1 – Surcoût d'exécution des contre-mesures.

contraire. La machine virtuelle peut ensuite vérifier que le tableau de bits correspond à une séquence calculée en dehors de la carte.

Double pile

La cinquième approche [CIC2] consiste à étendre le mécanisme classique de détection des dépassements (*overflow*) ou soubassements (*underflow*) de la pile java Card. Par défaut, pour chaque appel de méthode une frame est empilée sur la pile Java et contient les paramètres de la méthode suivie des variables locales utilisées dans le corps de la méthode. La dernière proposition est de séparer les éléments empilés en deux catégories : une pour les types primitifs et l'autre pour les références.

Afin d'optimiser l'utilisation mémoire, l'emplacement de la pile est décomposé en deux sous-piles. La pile pour les types primitifs est positionnée à partir de l'adresse basse de la zone mémoire et empile les types de manière ascendante. La pile pour les références est positionnée à partir de l'adresse haute et empile les références de manière descendante. Lors de l'exécution, les éléments sont empilés en fonction du type de l'instruction sur l'une ou l'autre pile.

4.3 Bilan

L'ensemble des idées présentées ci-dessous fonctionnent sur le même principe : conserver de l'information sur la sémantique du bytecode afin de la vérifier pendant l'exécution. La problématique est de trouver quelle est l'information la plus pertinente à extraire afin de conserver des bonnes propriétés de détection d'attaque et de performance (comme présenté dans la section 2.4) :

- le surcoût d'exécution : les vérifications proposées peuvent être coûteuses et rajouter un coût à l'exécution de chaque instruction mais aussi à certaines étapes ;
- le surcoût pour la carte en terme de mémoire :
  - EEPROM : égale à l'augmentation de la taille du fichier CAP (soit la taille du *Custom Component*) ;
  - ROM : l'accroissement du code de la machine virtuelle par l'implémentation de nos propositions.
- l'efficacité des contre-mesures : c'est-à-dire le taux de détection d'une attaque en faute ;
- la latence d'exécution : le nombre d'instructions exécutées entre une instruction ayant subi une faute et sa détection.

Le tableau 4.1 présente les performances propres (en dehors de l'efficacité) en implémentant chaque proposition en native sur SimpleRTJ (voir section 4.5). L'efficacité des contre-mesures et la latence de détection font partie du travail sur le simulateur de faute SmartCM (section 5.5).

L'ensemble de solutions est disponible pour les encarteurs qui veulent protéger leurs cartes contre des attaques en faute avec un impact raisonnable pour les ressources de la carte (inférieur à 5%). Les objectifs annoncés dans le chapitre 2 ont été atteints, notamment avec l'intégration au flot de développement Java Card.

De plus, la notion d'applet mutant (voir section 3.2) appelle à considérer des vérifications **pendant l'exécution**. En effet, dans le cas des attaques EMAN, nous possédons une grande précision temporelle et spatiale, mais nous avons besoin des clés de chargement et de l'absence de BCV. Une attaque en faute possède une moins bonne précision mais peut affecter une carte sans avoir de clés de chargement et peut corrompre un applet, même si celui-ci a été vérifié par un BCV. La précision croissante des attaques en faute et la diminution du prix des équipements rendent cette hypothèse réaliste.

Du point de vue du fondeur, la conséquence est de renforcer la sécurité de la machine Java Card. La solution extrême serait de construire une machine virtuelle défensive [91] afin de vérifier l'ensemble des propriétés vérifiées par un vérificateur de bytecode mais durant l'exécution. Elle doit alors implémenter de nombreux mécanismes pour apporter la même protection qu'offre un BCV. Nous pouvons citer par exemple :

— *runtime type checking* : La connaissance du type des éléments dans la pile afin d'éviter une attaque par confusion de type ;
— La vérification de la taille de la pile après chaque opcode, qui ne doit pas être négative ou supérieure à la limite de taille déclarée pour chaque méthode (max_stack) ;
— La vérification qu'une instruction de saut (conditionnelle ou directe) aboutit dans la même méthode où l'instruction est présente.

Mais le coût d'exécution d'une telle machine est prohibitif (surtout si la règle des 5% est conservée), notamment vis-à-vis des ressources limitées de la carte en terme de mémoire et de puissance de calcul. Nous pouvons supposer qu'une telle machine n'est pas déployée actuellement.

A noter qu'il existe des travaux réalisés par l'industrie sur ce problème mais dont il est difficile d'identifier l'efficacité et l'usage réel. Particulièrement, un brevet [48] propose une méthode utilisant la réécriture de code dont les auteurs annoncent un surcoût d'exécution nul.

Mais il est intéressant de remettre ces contre-mesures dans un contexte global de défense de la machine virtuelle. Une faute sur une carte peut avoir des impacts multiples : cryptographie, logique applicative, comportement de la machine virtuelle, contenu des secrets. La difficulté n'est donc pas de proposer des contre-mesures mais de garantir que l'ensemble des attaques soit couvert. A l'exception de la cryptographie proposant des réécritures d'algorithmes existants, la solution la plus universelle est d'exploiter la redondance d'exécution et/ou mémoire dans le microprocesseurs. Les dispositifs dédiées de redondance contre les fautes existent actuellement au niveau industriel et en production mais apportent un surcoût important (jusqu'à 90% de l'exécution), car c'est la méthode la plus efficace et générique contre des attaques connues voire inconnues, et le coût pour la carte reste constant même si de nouvelles classes d'attaques sont découvertes.

4.4 Mise à jour dynamique des applets

Un dispositif classique pour limiter l'impact de nouvelles menaces est de renouveler régulièrement le parc de cartes en circulation. C'est l'approche utilisée par le système bancaire qui limite la durée de vie des cartes à deux ans. Mais ce n'est pas le cas de tous

les secteurs d'activité : par exemple, les cartes SIM de téléphone n'ont aucune obligation de renouvellement. Le problème de renouvellement existe aussi dans le cadre d'ajout de support électronique aux documents officiels. Un exemple est celui des passeports électroniques dont la durée de validité est la même qu'un passeport classique, c'est-à-dire 10 ans. Dans ce contexte, il est difficile d'avoir des garanties fortes sur la résistance des dispositifs embarqués dans la carte sur une aussi longue période.

La *post-issuance* est le mécanisme permettant la mise à jour des applications dans une Java Card comme suit :

— arrêter puis détruire les instances en cours d'exécution de l'ancienne version de l'applet ;
— effacer l'ancienne version de l'applet ;
— charger la nouvelle version de l'applet ;
— créer les nouvelles instances de la nouvelle version de l'applet.

Mais il souffre de deux problèmes. Premièrement, le fait de détruire une ou plusieurs instances des classes attachées à un applet détruit aussi le contexte d'exécution de cet applet. Si la carte ne possède pas de dispositif adapté pour sauvegarder les états internes des instances pendant la mise à jour, ils sont perdus lors de la destruction des instances. La perte des états internes est aussi problématiques en cas de dépendance entre applets par le mécanisme de partage. Si un applet est renouvelé avec le mécanisme de *post-issuance* alors la cohérence des états internes entre les deux applets est compromise.

Deuxièmement, la mise à jour par *post-issuance* peut poser un problème de taille de transfert de l'applet complet. L'interface de communication entre la carte et son lecteur peut avoir une bande passante limitée, particulièrement dans le cas d'une carte sans contact dont la durée de communication est limitée. Il est donc difficile d'effectuer un renouvellement complet de l'applet. Si la mise à jour a une taille limitée par rapport à la taille de l'applet (correction de bug par exemple), alors il serait intéressant d'avoir une solution alternative pour ne mettre à jour qu'un sous-ensemble de l'applet.

Dans ce cadre, mes travaux se sont intéressés à la mise à jour dynamique des applications Java Card dans la carte. Cette problématique a été suggérée par notre partenaire Gemalto à partir de l'exemple du passeport électronique. Nous nous limitons dans ce travail à la mise à jour de applets Java Card (application chargées en *post-issuance* ou partie intégrante du système grâce à la *romization*) et non à la mise à jour du code natif.

Mise à jour dynamique

La mise à jour dynamique (*Dynamic Software Update* ou DSU) concerne la mise à jour d'instances de programme en cours d'exécution. Il n'existe pas de travaux sur la DSU dans Java Card mais il existe déjà plusieurs travaux de recherche sur la mise à jour dynamique de programme Java (et plusieurs réussites industrielles comme JRebel [7]), qui doivent répondre au besoin de flexibilité de la mise à jour, et plus précisément :

— Est ce que des changements non anticipés du programme sont possibles ?
— Est ce que des classes déjà chargées peuvent être modifiées ?
— Est ce que l'état du programme est conservé après la mise à jour ?

En dehors des solutions de support de debug (comme HotSwap [93]), de support de langage dynamique sur la machine virtuelle Java (comme Groovy [64] ou BeanShell [88]) ou de mise à jour de composant Jave nécessitant un redémarrage (comme Javeleon [47] utilisé par Netbeans ou OSGI Service Platform [89]), l'ensemble des approches de DSU pour Java se décomposent en deux catégories :

— les travaux modifiant la machine virtuelle : DVM [42], JDRUMS [100] et Jvolve [50] ;
— les travaux n'impliquant pas de modification de la machine virtuelle : DUSC [95], JavAdaptor [15] et Iguana/J [105].

Ces travaux montrent l'existence de quatre grands problèmes scientifiques : la mise à jour du code, la recherche des points sûrs, la mise à jour des instances et la préservation du système de type.

Mise à jour du code

La mise à jour du code consiste à réécrire partiellement le code binaire (*bytecode*) original et (éventuellement) gérer les différentes versions du code à mettre à jour (faire coexister ou non différentes versions du code).Une première solution présente dans JVOLVE est de réécrire dans un autre emplacement mémoire les séquences du code mises à jour avec un niveau de granularité au niveau classe. DVM utilise une stratégie équivalente en renommant les classes pour éviter une erreur du *classloader*.

Une autre solution est d'insérer à chaque début de méthode à mettre à jour une instruction de saut pointant vers le code de la méthode mise à jour. JDRUMS et Iguana/J utilisent une table d'indirection pour tous les appels de méthodes entraînant un surcoût à l'exécution de chaque appel de méthode modifiée ou non.

Dans le cas de solutions ne modifiant pas la machine virtuelle, il n'est pas possible de détruire des classes déjà instanciées avant modification. La seule exception étant JavAdaptor qui exploite le mécanisme d'introspection pour modifier le corps d'une méthode pendant l'exécution.

DUSC transforme l'application avant exécution pour être adapté au *swapping* de classe : chaque classe est décomposée en plusieurs sous-objets : l'état de la classe, l'interface de la classe et l'implémentation de la classe. Un *proxy* (appelé *wrapper*) est généré pour remplacer la classe originale et appeler les différents sous-objets (entraînant donc un surcoût d'exécution à chaque appel d'un membre d'une classe adaptée). Pendant l'exécution, seul le sous-objet implémentation est renouvelé.

Recherche de points sûrs

La recherche de points sûrs concerne la détection du moment opportun pour effectuer la mise à jour dynamique. Ces points sûrs doivent garantir que l'application de la mise à jour laisse le système dans un état cohérent. Dans le cas d'un programme Java, les références à mettre à jour se situent soit dans les instances, soit dans la pile de chaque *thread* Java. Le problème réside dans le fait qu'une méthode (ou un champ) modifié (ou enlevé) dans la mise à jour peut être référencé dans la pile Java. Il faut donc attendre le moment où ces références disparaissent de la pile.

Obtenir un point sûr consiste donc à attendre que les méthodes dites restreintes (les méthodes modifiées, présentes dans la pile et dont la mise à jour n'est applicable qu'à la fin de son exécution) aient fini leur exécution. Néanmoins, toutes les méthodes modifiées présentes dans la pile ne sont pas forcément toutes restreintes. De même, la mise à jour de leur code peut ne pas avoir d'impact sur l'exécution en cours (cette propriété doit alors être déterminée par une analyse *offline* ou *online* du contenu de la méthode).

La plus simple approche est de rejeter une mise à jour si une méthode restreinte est présente dans la pile d'exécution d'un *thread* Java. Par exemple DVM détruit le *thread* et une exception est levée.

Une approche utilisée dans l'ensemble des solutions modifiant la machine virtuelle est d'attendre que celle-ci atteigne un point sûr. JVolve considère par exemple que les points sûr pour une mise à jour sont un sous-ensemble des points sûrs de la machine virtuelle. Ces derniers sont des emplacements dans l'exécution de la machine virtuelle où s'exécute le ramasse-miettes et où les *threads* sont ordonnancés. A chaque fois que ce point est atteint, Jvolve vérifie que l'ensemble des piles de chaque *thread* ne contient pas de méthodes restreintes. Dans le cas contraire, il utilise un mécanisme de barrière pour chaque méthode restreinte afin de pouvoir intercepter la fin de chaque méthodes restreinte.

Les solutions sans modification de la machine virtuelle ne possèdent pas cette problématique : ils sont en général dans l'incapacité d'introspecter la pile Java. Les stratégies déployées s'appuient sur des notions de conteneurs ou de proxys modifiant le contenu interne de l'implémentation des classes de manière transparente, *i.e.* les références de chaque instance ne sont pas modifiées et donc n'ont pas besoin d'être modifiées dans les piles ou le tas.

Mise à jour des instances

La mise à jour des instances amène principalement la question de déduire le nouvel état de la mise à jour de l'instance. La logique de l'application et de sa mise à jour étant difficile à évaluer automatiquement, des fonctions de transfert existent afin de gérer les transitions entre l'ancien et le nouvel état. L'approche présente dans la majorité des solutions comme DUSC est de faire un duplication d'état un-à-un : l'état original de l'instance est copié (*shallow copy*) vers l'instance du nouvel objet. Si des mécanismes de migrations plus sophistiqués sont nécessaires, les fonctions de transfert sont soit fournis par le programmeur, soit des méthodes de transfert par défaut sont utilisés pour les nouveaux champs (cette dernière fonction est supportée par JDrums et JVolve).

Il existe deux approches pour déterminer le moment du transfert des états des instances. Le premier est "paresseux" et implique la mise à jour des instances lorsqu'elles sont accédés la première fois après la mise à jour, afin de réduire le temps d'exécution de la mise à jour. Par exemple JDrums et DVM interceptent les accès aux instances grâce à un niveau d'indirection supplémentaire entraînant un surcoût d'exécution pendant la suite de l'application. De son côté, JVolve s'appuie sur le ramasse-miettes pour identifier dans le tas l'ensemble des instances en cours d'exécution pendant la mise à jour. Lorsque le point sûr est atteint, le tas est parcouru afin de mettre à jour chaque instance mais aussi les références pointant vers les anciennes instances.

Les mécanismes de mise à jour avec modification de la machine virtuelle ne nécessitant pas de modification de la machine virtuelle, ils ne peuvent modifier des instances en cours d'exécution, et ne possèdent donc pas de méthode de transfert.

Préservation du système de type

Une autre problématique associée est celle de la préservation du système de type. Le plus simple est le changement du corps d'une méthode, supporté par l'ensemble des méthodes. Mais pour des cas plus complexes, il n'existe pas à notre connaissance de solutions avec une mise à jour consistante vis-à-vis de toutes les sémantiques possibles de mise à jour [15] et des travaux prouvent que ce problème est indécidable [110].

Les approches ne nécessitant pas de modification de la machine virtuelle ne sont pas transparents, elles introduisent des changements de la hiérarchie des classes par renom-

mage et encapsulation par *wrapper*. Dans le cas de DUSC et Iguana/J, il n'est pas possible d'ajouter ou enlever des membres publics d'une classe, ou encore de modifier la signature des membres publics, car les références de chaque classe ne peuvent être modifiées. Par l'usage de HotSwap pour introspecter la machine virtuelle, JavAdaptor est capable de partiellement mettre à jour la hiérarchie d'héritage.

Les solutions modifiant la machine virtuelle proposent une meilleure préservation du système de type, comme changer les signatures de méthodes (type, nombre d'arguments), ajouter ou enlever des membres statiques ou non de classe et leur modificateurs. Mais certaines opérations peuvent ne pas être disponibles, comme par exemple JVolve qui ne supporte pas la permutation de classe dans une hiérarchie de classe.

Approches

Durant mes travaux qui se sont déroulés pendant la thèse de doctorat de Agnès Noubissi [ST3, WI2, WI3, CI8, CNC2, CNC4, CN4, PO1, PO2], j'ai fais certains choix suite à l'environnement particulier des cartes à puce. Premièrement, une mise à jour doit réussir ou échouer mais laisser le système dans un état cohérent avec l'ancienne version. Deuxièmement, afin d'avoir un mécanisme simple, la granularité des mises à jour est celle de la méthode. Enfin, le cadre de notre application concerne les cartes à puce, donc avec des contraintes très fortes en terme de mémoire (EEPROM, ROM et RAM) et de calcul. C'est pour ces raisons que les choix techniques de notre mécanisme de mise à jour dynamique se doivent de soulager les ressources de manière efficace. Par exemple, la création de la mise à jour *offline* doit avoir comme objectif d'éviter de surcharger la carte avec des opérations complexes.

Dans le cas de la mise à jour du code, il n'est pas possible de faire coexister l'ancienne et la nouvelle version de chaque classe après la mise à jour sur la carte, la ressource EEPROM étant trop limitée. L'utilisation de fonctions d'indirection impose aussi un surcoût en termes de calcul (un appel de méthode d'une application aboutit à deux appels de méthodes dans la machine virtuelle) et de mémoire (utilisée par la table d'association "appel de méthode" vers "adresse de la méthode à jour"). De même, l'insertion d'une instruction de saut au début de chaque méthode de l'ancienne version n'est pas possible à cause de la limitation des instructions de saut Java Card qui sont limitées à un saut relatif de plus ou moins 127 octets.

La solution choisie pour la mise à jour du code est d'utiliser une réécriture par copie / modification suivie d'une édition dynamique des liens. Durant le chargement, une copie est effectuée de l'ancienne version avec une modification des éléments changés. Afin d'éviter tout calcul supplémentaire à la carte, cette information est directement disponible dans le fichier de mise à jour transmis à la carte.

En ce qui concerne la mise à jour des données, les objectifs d'optimisation des ressources et d'atomicité de la mise à jour nous imposent d'effectuer la mise à jour en une seule fois. Une approche paresseuse de mise à jour des données n'est donc pas possible. La séquence de mise à jour doit donc inclure la création des champs statiques et leur initialisation par la méthode `clinit`, puis la création de tout nouvel objet et leur initialisation par la méthode `init`. Ces étapes sont suivies par le parcours du code, de la pile et des instances dans le tas afin de détecter et modifier toutes références pointant sur des anciennes données vers la nouvelle version.

Concernant la nécessité des méthodes de transfert, le programmeur a la charge de les écrire explicitement. En effet, il est difficile de déterminer automatiquement les nouvelles

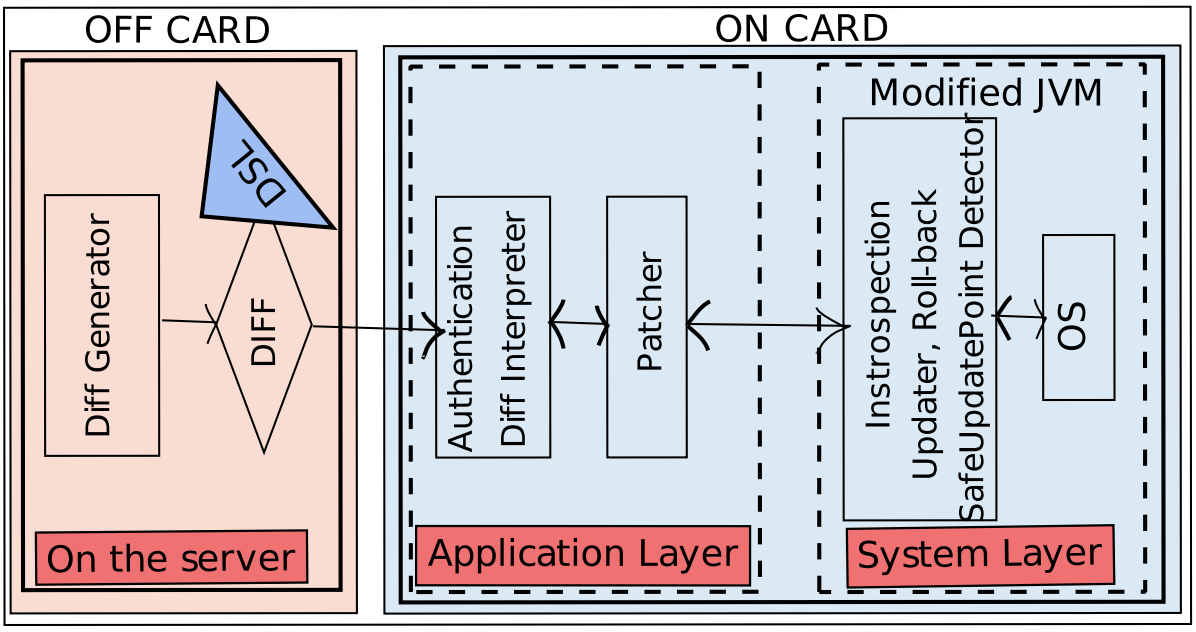


FIGURE 4.2 – Architecture de EmbedDSU

valeurs. Par exemple lors d'un changement d'un algorithme cryptographique (cas typique de la mise à jour d'une application dans la carte), comment le mécanisme de mise à jour peut établir les changements induits sur les clés et autres valeurs est difficilement identifiable de manière automatique par la complexité des effets de bords comme les attaques par canal auxiliaire par exemple.

En ce qui concerne la détermination des points sûrs, la politique choisie est de surveiller toutes les *frames* dans la pile équivalentes à une méthode restreinte. Lors du déclenchement de la mise à jour, un comptage des *frames* appartenant à des méthodes restreintes est effectué, puis à la fin de l'exécution de chaque méthode le compteur est décrémenté si la *frame* appartient à une méthode restreinte. Lorsque le nombre de *frames* de méthodes restreintes atteint zéro, la mise à jour est appliquée.

Cet algorithme simple possède deux défauts. Premièrement, il n'est pas capable de gérer le problème des boucles infinies dans le code. Mais dans le cadre du développement Java Card, ce n'est pas une structure de code recommandée pour l'écriture d'un applet. De plus, il est tout à fait possible d'ajouter une phase d'analyse statique des applets avant chargement pour détecter la présence possible de ce motif. Deuxièmement, nous n'avons pas de garantie sur l'atteignabilité, particulièrement dans le cas d'une machine virtuelle conçue pour fonctionner en permanence (une machine virtuelle Java Card est instanciée avant l'émission de la carte et ne s'arrête jamais). Je considère néanmoins que la solution proposée répond aux problématiques au regard des ressources contraintes de la carte (développer de meilleurs mécanismes de recherche de point sûr avec des propriétés plus fortes risque d'avoir une complexité et un usage des ressources plus important).

EmbedDSU

EmbedDSU (pour Embeded Dynamic Software Update) est un ensemble d'outils et de modifications d'une machine virtuelle Java Card permettant la mise à jour dynamique des composants Java. Il est composé de deux grandes parties présentées dans l'Illustration 4.2 :

— une partie off-card qui génère un fichier DIFF à partir des deux versions des classes et des fonctions de transfert ;
— une partie on-card qui charge le fichier DIFF et applique les modifications à l'application.

Off-card

Le générateur de DIFF détecte entre deux versions d'un groupe de classes les différences suivantes :
— le constant pool (ajout, suppression ou modification d'une entrée) ;
— les droits d'accès à la classe ;
— les champs de la classe (valeur initiale, droits d'accès, index dans le constant pool) ;
— les méthodes de la classe (droits d'accès, signature, variables, locales et instructions).

Dans un premier temps, l'outil étudie les tables de constantes (*constant pool*), les droits d'accès, les champs et méthodes de classe afin de déterminer les éléments supprimés, ajoutés et modifiés. Durant la seconde phase, le générateur de DIFF calcule le CFG de chaque méthode et en déduit les modifications sur les instructions. En effet, l'ajout ou le retrait d'appel de méthodes dans le corps d'une méthode peut entraîner des modifications du constant pool. En fonction des modifications engendrées, le constant pool est réorganisé et une renumérotation des index est effectuée, afin de savoir sur quel index du constant pool sont appliquées les modifications des champs et des méthodes embarquées dans la carte.

L'outil de génération de DIFF accepte des fonctions de transfert dynamique écrites par le programmeur pour l'initialisation des champs statiques ou pour la mise à jour de la valeur de variables. Pour spécifier les changements, le programmeur fournit une classe possédant une méthode statique prenant en entrée l'ancienne et la nouvelle instance de la classe. De même, il est possible d'avoir des fonctions de transfert statiques pour les champs ajoutés : soit le programmeur spécifie la valeur, soit une valeur par défaut est générée par l'outil.

Le mécanisme de génération de DIFF accepte les modifications suivantes :
— la modification de la signature d'une classe : ajout et suppression de méthodes et de champs, modification de la signature d'une méthode ou d'un champ (type et droits d'accès), modification de la valeur d'initialisation d'un champ ;
— la modification de la signature des méthodes : ajout et suppression d'un paramètre, modification du type d'un paramètre ou de la valeur retournée, modification des accesseurs d'une méthode ;
— la modification du contenu des méthodes : ajout et suppression d'une variable locale, modification de la signature d'une variable locale, ajout et suppression d'une ou plusieurs instructions.

On-card

EmbedDSU étend la machine virtuelle avec un ensemble de modules. Lors d'une mise à jour, le fichier DIFF est lu par un interpréteur afin de connaître les structures à mettre à jour. Ensuite, un mécanisme d'introspection de la VM détermine les instances de classes à mettre à jour et les *frames* associées aux méthodes restreintes.

La machine virtuelle continue à s'exécuter tant qu'elle n'a pas atteint un point sûr (c'est-à-dire s'il existe encore des méthodes restreintes dans la pile). Pour avoir un surcoût

faible à cette opération lors de la mise à jour, le nombre de références dans la pile à des méthodes restreintes est calculé puis décrémenté à la fin de l'exécution de chaque méthode restreinte. Une fois atteint zéro, la mise à jour est appliquée. Le temps d'attente de ce point sûr est limité par le système afin de garantir que la machine virtuelle puisse annuler la mise à jour en cas de non-atteignabilité du point sûr.

Une fois le point sûr atteint, la mise à jour est appliquée :

— un mécanisme de recopie (*modify-on-copy*) copie le *bytecode* original de chaque méthode modifiée vers un emplacement libre en EEPROM en le modifiant à la volée à partir des instructions du fichier DIFF ;
— les instances sont aussi dupliquées avec le même mécanisme de recopie en ajoutant, modifiant ou supprimant les champs dans la nouvelle instance ;
— les références vers les anciennes instances de classes sont recherchées dans les *frames* présentes dans la pile Java Card et remplacées par les références vers les nouvelles instances de classes ;
— Les adresses de retour présentes dans les *frames* sont aussi remplacées vers l'adresse de ces méthodes mises à jour ;
— la signature des classes dépendantes d'une classe mise à jour (c'est-à-dire contenant des appels ou accédant aux champs d'une classes mise à jour) est actualisée afin de référencer vers la nouvelle version de la classe ;
— enfin, les transferts d'états sont exécutés.

Afin d'éviter de mettre la carte dans un état indéterminé dû à une erreur de mise à jour, deux approches complémentaires sont présentes. Le premier est la nature incrémentale du processus de mise à jour, c'est-à-dire qu'il existe toujours une ancienne version de chaque classe en mémoire de la carte tant que la mise à jour n'est pas complètement réalisée. Cette idée s'appuie sur l'usage de la copie en modification pour la mise à jour du code et des instances. Le second est la présence d'un mécanisme de *rollback* capable de remettre en place les anciennes versions des classes et des instances en cas d'erreur, d'opération illicite ou de non atomicité de la mise à jour (cas de l'arrachage de la carte par exemple).

Bilan

Le travail développé dans cette section concerne la démonstration qu'un mécanisme de mise à jour dynamique d'applications Java Card est possible tout en respectant les contraintes de la carte. Au niveau mémoire, le surcoût est limité à l'ajout du nouveau *bytecode* à l'EEPROM avec une granularité de la méthode remplacée. Afin de garantir une mise à jour cohérente, la recherche du point sûr n'a pas d'impact sur les références dans la pile qui font partie de l'exécution et se fait en un temps limité (le système retourne dans un état stable grâce au mécanisme de *rollback* en cas de problème).

Un des objectifs de ce travail est d'avoir un système simple afin de pouvoir certifier l'ensemble du processus de mise à jour dynamique. Par exemple, les modifications sont limitées à l'ajout de code (classes, membres variables) ou des transferts d'exécution à de nouvelles méthodes afin de réduire les modifications possibles du code Java entre deux versions pour éviter les ambiguïtés lors de la création du DIFF. De plus, chaque mécanisme implémenté dans la carte (*rollback*, détection de point sûr, etc.) est de taille légère avec une complexité algorithmique limitée.

Il existe quelques limitations à cette proposition. Premièrement, EmbedDSU ne supporte pas la modification des interfaces et de la hiérarchie d'une classe, car l'impact pour vérifier et générer les informations de mises à jour est trop important pour être couvert

dans notre travail. Dans ce cas, il nous faut considérer les méthodes et variables héritées pour connaître leur nouvelle position dans l'arbre d'héritage et sur l'ensemble des objets hérités, que ce soit du point de vue de l'analyse off-card mais aussi de l'impact dans la carte. Mais nous considérons que l'objectif de notre solution concerne des modifications légères et en aucun cas, une refonte complète de l'architecture de l'application à mettre à jour.

De plus, il manque une preuve formelle de la validité de notre approche. Les choix techniques et algorithmiques sont simples mais nous n'avons pas de preuve formelle sur le fait que le passage de l'ancien état au nouvel état se fasse correctement. Plus généralement, il serait intéressant de relier ce travail au support du BCV, afin de montrer qu'un ancien fichier CAP valide d'un point de vue d'un BCV plus les différents DIFF générés est égal au fichier CAP de la nouvelle version validée par un BCV, permettant d'éviter d'embarquer un BCV *oncard* qui serait nécessaire pour chaque mise à jour.

4.5 Ressources

Expérimenter la machine virtuelle Java Card

Il n'existait pas à l'époque de nos travaux d'implémentation de la JCVM disponible pour nos travaux (sous licence open-source par exemple). Il n'existait donc aucun moyen de pouvoir travailler sur la machine en elle-même. Même si les attaques EMAN ont permis d'avoir un aperçu sur le code natif de la JCVM, il paraît difficile de pouvoir lui ajouter des modifications.

Développer une machine Java Card complète est en dehors de nos ressources humaines (particulièrement à l'époque de nos travaux, où notre équipe était en émergence), les ressources de développement serait trop importante, notamment par la difficulté d'implémenter correctement certains mécanismes comme les transactions ou l'API complète Java Card.

Une alternative serait de construire une communauté de recherche autour d'une implémentation open-source. Mais la technologie Java Card fonctionne sur le principe de licence (coûteuse) auprès de fabricants. La conception et la disponibilité d'une machine open-source représente un risque légal pour les développeurs car cette implémentation pourrait représenter une alternative par les encarteurs.

J'ai donc cherché une alternative pour expérimenter des travaux de protection et de modification de la machine virtuelle Java Card. Il est apparu qu'il n'est pas nécessaire d'avoir une implémentation complète de la norme Java Card. Au regard des travaux exposés dans ce chapitre, l'intérêt est de premièrement pouvoir vérifier l'exécution du code par rapport à une structure de données chargée simultanément, et deuxièmement de pouvoir introspecter la machine virtuelle et modifier des données associées pendant son exécution. Les propriétés recherchées sont :

- une machine virtuelle proche du modèle Java et/ou Java Card, c'est-à-dire avec le même principe de fonctionnement (machine à pile) et une logique associée d'instructions proche de ce modèle ;
- une machine virtuelle respectant les contraintes des systèmes embarqués, pour avoir une environnement probant lors de la collecte de métriques pour les résultats d'évaluation ;
- la capacité de pouvoir écrire nos propres programmes Java et de pouvoir les exécuter sur notre plate-forme, avec un processus de conversion des classes vers le

format exécutable par la machine

On notera que les fonctions associées du JCRE ne nous intéressent pas dans nos travaux : chargement et déchargement *post-issuance* d'applications (notre modèle de mise à jour est dynamique et ne nécessite pas l'arrêt et le redémarrage des instances), le pare-feu (aucun de nos travaux n'a d'interaction avec cette entité), la notion de contexte d'application et plus généralement l'API Java Card.

Le choix s'est porté sur Simple Real TIme Java (SimpleRTJ) [1], une machine virtuelle Java pour des microprocesseurs 8 bits et 16 bits (même cible que les microprocesseurs de carte à puce) avec la gestion de modèle mémoire en banc (*banked*) ou linéaire (*linear*) et une empreinte mémoire faible (moins de 24Kb). Elle supporte les principales fonctionnalités de la machine virtuelle Java, comme le ramasse-miettes et fonctionne de manière identique (pile, tas, élément indexé dans le constant pool).

Le modèle de chargement est légèrement différent à celui de la Java Card : comme le modèle de développement de SimpleRTJ considère l'existence d'une seule application dans le système (contrairement à Java Card acceptant plusieurs applets simultanément) l'édition des liens de l'application avec les bibliothèques Java est réalisée en dehors du système (hors-ligne), et produit un seul binaire contenant l'ensemble de l'application et ses dépendances de la bibliothèque Java de SimpleRTJ. Nos travaux étant appliqués sur un code déjà chargé et exécuté, nous pouvons considérer que cette différence n'a pas d'impact sur notre évaluation.

Il n'existe aucune optimisation dans l'implémentation de cette machine, elle traite une par une les instructions Java de l'application sans l'aide de mécanisme de type Just-in-Time (JIT), ce qui est identique au modèle d'exécution des instances de Java Card que nous avons étudié. De plus, le jeu d'instruction est identique à celui de Java, donc très proche de celui de Java Card. Nous considérons donc que les métriques collectées pendant les expérimentations sont tout à fait pertinents. la seule différence réelle est que notre prototype de SimpleRTJ s'exécute uniquement en RAM. Le surcoût durant la mise à jour due aux écritures importantes dans l'EEPROM serait une évaluation à faire pour considérer notre solution.

SimpleRTJ possède son propre format de fichier non documenté ; les sources du *classlinker* permettant de générer le fichier exécutable ne sont pas disponibles. J'ai développé notre propre *classlinker* en utilisant des méthodes de rétro-conception pour comprendre l'organisation du fichier exécutable. Pour des raisons de restrictions dans la licence de SimpleRTK, le code source n'est pas distribuable.

Participants

Les travaux présentés dans cette section ont été réalisés durant la thèse de doctorat d'Ahmadou Séré [ST4, JIC1, CIC6, WI4, CI6, CNC5, CN5] pour la partie contre-mesures d'attaques en faute, avec le travail de Jean Dubreuil sur l'implémentation de la double pile [CIC2]. Pour la partie mise à jour dynamique, j'ai co-encadré le travail de thèse de Agnès Noubissi [ST3, WI2, WI3, CI8, CNC2, CNC4, CN4, PO1, PO2] qui fut accompagnée par Josselin Dolhen pour le développement d'un outil de génération de DIFF et de Keita Ansoumane pour l'implémentation d'un serveur de mise à jour.

1. Il semble que la page originale du projet http://www.rtjcom.com/ ne soit plus disponible, une version miroir est disponible à l'adresse http://www.csc.uvic.ca/~mcheng/360/notes/simpleRTJ/

Chapitre 5

De la défense des applications par analyse statique

5.1 Introduction

Les attaques contre les Java Card exposées dans la problématique (voir chapitre 2) et celles découvertes durant mes travaux (voir chapitre 3) sont des menaces contre lesquelles les encarteurs vont proposer des contre-mesures. Par exemple une approche exposée dans le chapitre 4 est de modifier la machine virtuelle pour la défendre contre les fautes.

Ces modifications sont dépendantes de l'encarteur et concernent le système. Elles sont donc invisibles pour les développeurs, les intégrateurs et les utilisateurs finaux. Mais il est possible que ces contre-mesures soient absentes pour des questions techniques (cartes *legacy* ou non adaptées) ou commerciales (segmentation du marché), d'autant plus que rien ne prouve la présence d'une contre-mesure (sauf pour un niveau de certification important). De plus, il existe des faiblesses au niveau applicatif, c'est-à-dire ne pouvant être contrecarrées par la machine virtuelle ou plus généralement la plate-forme d'exécution.

Dans ces deux cas, la responsabilité incombe à l'intégrateur et/ou le prestataire de service de s'assurer de la sécurité des applications produites. Mais cet objectif a deux contraintes majeures :

— Le développement d'applications pour carte étant facilité par l'usage de Java comme langage de programmation, une importante concurrence entre prestataire de service diminue le prix du développement mais aussi l'attention dédiée par les programmeurs à la qualité du code produit.
— L'impossibilité de connaître les spécifications internes de la carte rend difficile la conception d'un code sécurisé, par exemple contre les attaques en faute.

Pour répondre à ce problème, il existe dans l'écosystème plusieurs entités en charge de la certification des applications (en général conjointement au support physique). Mais les services de ces entités peuvent se révéler coûteux (ralentissant le déploiement de l'écosystème) et dépendants des encarteurs.

Dans ce chapitre sont présentés plusieurs cas où la responsabilité des secrets dans la carte est à la charge du développeur d'application Java Card, voire partagée entre l'intégrateur et le(s) développeur(s). Plusieurs approches d'analyse de programmes sont alors présentées afin de renforcer la sécurité des programmes contre des menaces particulières.

5.2 De l'analyse d'applet

Par défaut, plusieurs mécanismes de Java Card vérifient des propriétés de sécurité de l'applet avant et après chargement dans la carte. Durant la compilation d'un applet, un certain nombre de vérifications sont effectuées comme la sémantique du code, le typage des objets, l'héritage, les symboles présents dans les *imports*, etc. Puis le vérificateur de *bytecode* (externe ou interne à la carte) contrôle que l'applet est cohérent : les références internes de l'applet sont présentes, la pile a toujours la même taille au début et à la fin de l'exécution d'une méthode, etc. Enfin, le pare-feu garantit pendant l'exécution l'isolation de l'applet. Ces composants garantissent l'intégrité et l'isolation de la machine virtuelle en présence d'un code malicieux voulant échapper à l'isolation de la machine, mais il existe d'autres catégories de menaces sur une application.

Le développement d'une application nécessite d'empêcher que des secrets soient utilisés de manière inconsistante. Les API Java Card proposent des fonctions permettant de simplifier des opérations critiques (saisie d'un PIN par exemple) mais le développeur peut ré-implémenter lui-même de manière non sécurisée certaines fonctionnalités déjà présentes. De plus, la sémantique de l'API ou la découverte de nouvelles failles peut amener à prendre des précautions quant à son usage. Enfin, l'évolution des menaces nécessite d'amener les bons moyens de les contrecarrer aux développeurs et prestataires afin d'éviter le coût important d'une certification.

Si nous cherchons à proposer des solutions pour répondre à ces problématiques, il faut qu'elles soient compatibles avec l'écosystème, c'est-à-dire qu'il faut disposer de mécanismes disponibles avant chargement dans la carte. Une classe de solutions est contenue dans l'analyse statique de programme, même si cette approche est limitée par le théorème de Rice démontrant que toute propriété non triviale d'un langage de programmation Turing complet est indécidable [114]. Malgré cette limitation, c'est cet ensemble de méthodes que nous avons appliqué au problème de vérification des programmes avant chargement.

Un exemple de problème impliquant les programmeurs et/ou l'intégrateur est de garantir que tous les échanges entre applets et vers l'extérieur de la carte répondent à une politique de sécurité donnée, c'est à dire que la circulation des données dans une carte doit garantir des propriétés pour empêcher un secret d'être accessible. Pour répondre à cet objectif, l'analyse de flots est une méthode consistant à déduire l'ensemble des valeurs possibles d'une variable durant l'exécution d'un programme à partir de son graphe de flot de contrôle (CFG). Plusieurs propositions ont été faites au niveau Java pour l'implémentation d'un tel mécanisme. On peut citer par exemple JBlare [27], un IDS encapsulé dans une machine virtuelle Java capable de vérifier une politique de sécurité sur les flots d'un programme Java.

Un autre objectif est de vérifier des propriétés d'un modèle sur le code avec des méthodes de model checking. Ces propriétés sont donc plus fortes que vérifier les flots d'information, car il s'agit de vérifier que des propriétés sont vraies sur l'ensemble d'un (modèle du) programme. Les assistants de preuve permettent à l'utilisateur de construire et vérifier une preuve vérifiée sur un modèle. EST/Java, par exemple, permet l'analyse de programme Java et utilise un prouveur de théorème pour vérifier que l'implémentation vérifie la spécification, décrite sous forme d'annotations Java représentant des invariants de classe, et des pré-conditions et post-conditions d'une méthode.

Concernant Java Card, l'analyse statique d'applet a déjà été appliquée dans le cas de la Java Card pour l'analyse avant chargement dans la carte. Le langage de spécification JML [71] pour Java peut-être utilisé par des assistants de preuves comme Jack [54] ou

Loop [97]. Le projet PACap [98] propose de vérifier si une application est correctement implémentée et respecte une politique de sécurité qui définit les partages autorisés. Mais ces outils vérifient des propriétés avant chargement et ne sont pas adaptés à l'environnement ouvert de la Java Card.

Pour répondre à cette problématique de plate-forme ouverte, une alternative à l'analyse statique pour l'analyse de flot est STAN [55], un outil d'analyse dynamique ("on-card") capable de s'assurer dynamiquement que les flots d'information respectent une politique de sécurité définie par l'étiquette associée à chaque champ d'une classe (secret ou public).

Un autre travail est de vérifier les temps attendus d'exécution maximale dans une carte. Gilles Grimaud et al. ont proposé d'étendre l'exokernel CAMILLE [103] pour la définition d'un ordonnanceur collaboratif avec les contraintes d'une carte à puce. Leur travaux [77], appartenant à la classe de l'analyse statique, s'intéressent à extraire en dehors de la carte les calculs importants afin de pouvoir estimer "on-card" le WCET du code en cours d'exécution par la carte et les ordonnanceurs embarqués.

La machine virtuelle ne peut pas résoudre par son design ces deux problèmes : les outils servent à apporter des garanties au programmeur de l'application, d'une part, et que les flots de contrôle soient vérifiés et que le WCET soient garantis d'autre part.

5.3 Protection du code du programmeur

À la suite de la sortie du guide de référence AFSCM [9] et d'un besoin de réduire le coût de la certification des applets NFC (aussi appelés "cardlets"), mes travaux se sont concentrés sur la construction d'un outil d'analyse statique vis-à-vis des bonnes pratiques recommandées dans le guide.

Vérifier ces bonnes pratiques signifie chercher dans le code la présence ou l'absence de motifs particuliers. Il existe de nombreux travaux pour la recherche de bug très particulier comme les *race conditions* [68]. A l'opposé, il existe des vérificateurs de code comme Jlint [1] ou PMD [2] qui explorent le code au niveau source ou bytecode afin d'identifier des erreurs de programmation, comme celles décrites dans [51]. Notre travail s'est construit autour de FindBugs [78], un programme développé pour être simple [3] et extensible afin de chercher des motifs de code particuliers. Pour chaque classe soumise, le logiciel exécute une portion de code décrivant un bug avec une construction lorsque nécessaire du CFG ou le flot de données à partir du code étudié. Pour des raisons de performance, Findbugs ne supporte pas l'analyse inter-procédurale entre classes ni l'analyse inter-classe car les motifs identifiés sont en général des appels à des API de Java et non des dépendances entre plusieurs classes écrites par le développeur.

Le développement pour FindBugs que j'ai dirigé dans le cadre de ce travail supporte l'analyse intra-procédurale (suivi de la propagation des flux de données internes à une méthode en tenant compte des différents types d'instruction, des branchements et des exceptions) et inter-procédurale (suivi de flux de données entre fonctions de l'application à partir des paramètres, des valeurs de retour et des champs communs ou méthodes d'une classe) de fichier classes. Une telle profondeur d'analyse est nécessaire car il faut un suivi

1. http://jlint.sourceforge.net/
2. http://pmd.sourceforge.net/
3. les auteurs de FindBugs indiquent que *"None of the detectors make use of analysis techniques more sophisticated than what might be taught in an undergraduate compiler course."*

de toutes les entrées mémorisées dans une variable (donc étiquetées) à travers les appels de méthodes, et ceci jusqu'à l'état final.

Les motifs recherchés appartiennent à l'une des catégories suivantes :

— Précédence d'appel de méthode à partir du CFG ;
— Présence ou non d'opérations (appel de méthode ou exception) dans le CFG à partir d'une méthode ;
— Appel ou non de méthodes statiques ou non d'une classe ;
— Valeur recommandée ou interdite pour le paramètre d'une méthode, nécessitant de connaître les valeurs possibles de chaque paramètre à l'aide d'une analyse de flot ;
— Détection de motif de code comme les boucles infinies ou le code mort ;
— Détermination du type / nom d'une variable à partir des informations de classes.

Bilan

L'analyse statique présentée dans ces travaux n'est pas d'une grande sophistication comparée à la littérature existante et réutilise de nombreux algorithmes développés dans le cadre de FindBugs. Néanmoins, il est intéressant de voir que ces travaux qui seront étendus par la spin-off de notre équipe Arya Security couvrent une grande partie des règles AFSCM.

Bien sûr, il existe des règles contractuelles qui sont inaccessibles à l'analyse statique : présence de documents de spécification, documentation explicite, obligation de validation par BCV vis-à-vis de l'intégrateur, etc. De plus, des décisions sont prises concernant des règles floues. Par exemple, limiter l'usage des ressources mémoires nécessite de suivre l'intégralité des allocations dans le tas et de lever un avertissement à partir d'un certain seuil.

Néanmoins, par rapport à d'autres approches d'analyses statique, nous sommes dans une position où nous validons les applets et où le programmeur a l'obligation d'en tenir compte. Ainsi, la présence de faux négatifs est extrêmement limitée car le programmeur va chercher à s'y conformer pour accélérer la phase de validation. C'est donc au programmeur de respecter les règles et non au programme d'analyse d'identifier leur validité réelle.

5.4 Défense des applications web contre les attaques XSS

Problématique

L'arrivée de nouvelles générations de carte à puce embarquant un serveur web oblige à repenser toute la sécurité de ces dispositifs par rapport au modèle traditionnel des cartes à puce. L'utilisation de cette technologie répond à une demande du marché pour l'intégration de ces technologies dans les systèmes d'information et les nouveaux périphériques mobiles. Par exemple, nous pouvons citer l'utilisation d'un butineur s'exécutant sur le mobile capable d'adresser et de demander des ressources directement à la puce. Un autre exemple est l'intégration d'une carte à puce dans un réseau afin que le système d'information puisse communiquer à l'aide de web services. En terme d'intégration et de performance, des serveurs web embarqués comme SMEWS [36] ont montré qu'il était

possible de concevoir des petits objets dans un environnement beaucoup plus contraint que la carte à puce.

L'analyse statique est l'une des solutions pour qualifier partiellement si l'implémentation suit un ensemble de bonnes pratiques. Mes travaux avec Arya Security sur l'analyse statique basée sur des règles AFSCM ont montré la validité de l'approche. La question intéressante serait de voir si ces travaux d'analyse statique pourraient être étendus sur la vérification d'applets web, comme par exemple des contrôles sur les bons usages de cookies et sessions, les détections d'attaques de type CSRF (Cross-Site Request Forgery), si l'usage de SSL avec HTTPS suit les recommandations de la configuration des ciphers dans ce cadre, etc.

L'attaque XSS (ou Cross Site Scripting) consiste pour un attaquant à injecter du code JavaScript malicieux dans certains champs non filtrés d'une application. Ce code est ensuite exécuté dans le navigateur de la victime si celle-ci charge une ressource de l'application intégrant ce code sans le nettoyer. Le code malicieux peut alors transférer des données accessibles par le code depuis JavaScript (cookie par exemple), modifier le comportement local de l'application (c'est-à-dire dans le navigateur de la victime), ou encore provoquer une indisponibilité de services.

Le danger pour un développeur d'applications pour Java Card Web Server est de développer une application possédant une faille XSS et donc casser le lien de confiance entre le navigateur web de l'utilisateur car si un attaquant mémorise dans la carte une séquence de caractères interprétées par un autre utilisateur comme une séquence javascript légitime, alors l'attaquant peut exécuter du code pour voler les cookies de sessions, rediriger l'utilisateur vers un autre site identique, ou encore exploiter des vulnérabilités sur le navigateur de la victime.

Des solutions de contre-mesures sans toucher à l'application existent à plusieurs niveaux :

— Une première classe de solutions est l'intégration de mécanismes de détection de code malicieux au niveau serveur (pare-feux applicatifs comme ModSecurity [4]). Ils possèdent l'avantage d'être indépendants de l'application, mais il est difficile de différencier le code malicieux du code JavaScript classique. De plus, ces services ne détectent que des séquences connues comme hostiles, ce qui est aisément contournable en réécrivant le code malicieux pour l'offusquer .
— Une deuxième solution est un mécanisme à l'intérieur du navigateur autorisant ou non l'exécution du code JavaScript. La décision de bloquer un code JavaScript se fonde sur la détection de séquences particulières ou l'utilisation de listes blanches et listes noires. Outre le fait que la détection de séquences est contournable, la décision de classifier un code comme potentiellement dangereux est de la responsabilité de l'utilisateur. De plus, il n'y aucune garantie que ce mécanisme soit présent ou actif dans un navigateur discutant avec la carte. Plus généralement, il est préférable de prendre le postulat que le navigateur de l'utilisateur n'offre aucune garantie.
— La dernière solution possible est le renforcement de la sécurité des applications serveurs par de bonnes pratiques et l'encodage des réponses retournées à l'utilisateur, ce qui demande une attention dédiée de la part du programmeur.

4. https://www.modsecurity.org/

Solution

Plus généralement, détecter un XSS est un problème difficile, car la logique de détection du paramètre dépend de ce qu'attend le programmeur et de l'emplacement du code malicieux dans la page HTML générée, sans compter le fait que le code JavaScript ne s'exécutera pas de la même façon en fonction du navigateur ou de l'endroit dans la page web où est réalisée l'injection.

A partir des faiblesses constatées de plusieurs mécanismes de filtrage XSS [31], plusieurs travaux ont émis l'idée de déterminer le type de chaque variable et de son usage dans la page HTML afin d'identifier au mieux la fonction à appliquer [49, 44]. La méthode générale est de teinter le contenu des variables afin d'identifier celles pouvant être retournées vers l'utilisateur et d'y appliquer un filtrage ou alors de placer de manière aléatoire mais consistante les tags HTML [22]. Mais les solutions proposées sont développées pour être utilisées de manière dynamique ou nécessitant un travail important de la part du développeur (par l'usage de patron pour générer le code HTML par exemple).

Mon travail s'est orienté alors sur une solution combinant analyse statique pour détecter la correcte utilisation de méthode de confiance [CI5]. Le support de ces méthodes d'analyse s'est construit autour de FindBugs afin de permettre la propagation d'étiquettes à partir des entrées de l'application jusqu'au potentiel sortie (affichage dans la page HTML retournée). Ainsi, la solution se compose de deux outils :

— un outil d'analyse statique, baptisé XSSDetector effectuant une analyse de flot pour vérifier que chaque paramètre d'entrée atteint une fonction de filtrage de confiance avant d'être retournée à l'utilisateur ;
— une API nommée JCXSSFilter permettant de nettoyer une variable contre les attaques XSS.

Un programmeur décidant d'utiliser nos outils pour développer son application utilise les fonctions de JCXSSFilter afin de nettoyer toute entrée dans son programme. Puis, une fois l'applet réalisé, il exécute l'outil d'analyse statique afin de vérifier que toutes les entrées du programme sont bien nettoyées avant d'être retournées à l'utilisateur.

Comme il n'existe pas de mécanismes standards en Java pour vérifier et neutraliser des données potentiellement malicieuses contre les XSS, la doctorante Nassima Kamel a développé le portage sur Java Card 3 de l'API proposée par OWASP appelée ESAPI [5]. Cette bibliothèque est reconnue comme fiable et possède de nombreux portages pour différents langages. Mais elle ne peut être chargée dans une Java Card 3 car elle est de taille importante (incompatible avec les contraintes de la carte) et utilise des structures de données non supportées par la Java Card 3 (HashMap, LinkedList, etc.). Le portage développé de cette bibliothèque a omis de nombreuses fonctions qui n'ont pas d'intérêt dans le cadre du projet (filtrage contre les injections SQL, par exemple).

Bilan

La solution proposée se décompose en 2 approches :

— un ensemble de fonctions dites "de confiance", c'est-à-dire capable de filtrer efficacement une variable contenant une entrée ;
— un mécanisme capable de détecter si cette méthode de confiance est bien appelée sur une entrée durant l'exécution de l'applet web en effectuant la technique de dépendance causale.

5. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

Ici encore, l'approche prise par mes travaux est d'abord de considérer des solutions déportant au maximum les opérations complexes en dehors de la carte. De plus, reprendre l'approche proposée dans la précédente section : apporter une garantie forte au prix d'une restriction de la liberté du programmeur. L'idée est de mettre à disposition du programmeur des méthodes pouvant nettoyer les entrées d'une application (afin d'éviter d'analyser l'intérieur des chaînes de caractères) et de vérifier leur présence.

Cette approche a un sens dans le contexte de l'écosystème, car elle réduit la certification à la vérification des méthodes de filtrage (et de leur éventuel ajout dans la carte par l'encarteur). La complexité de vérifier le filtrage est donc géré en dehors de l'écosystème de production des applets Java Card. La solution une fois posée est donc simple et facilement intégrable pour le prestataire de service et l'intégrateur.

5.5 Protection contre les attaques en fautes

Dans la section 4.2 est présenté un mécanisme pour la JCVM permettant de limiter l'impact des attaques en faute. Nous proposons ici de mettre en place un mécanisme vérifiant de manière statique (en dehors de la carte) les potentiels impacts d'une faute sur un *bytecode* donné afin d'aider le développeur à construire une application naturellement résistante aux attaques en faute.

Il n'existe pas de moyens pour un développeur ou un intégrateur de vérifier qu'un applet est résistant à une attaque en faute. L'utilisation d'une méthode de tests physiques [84] n'apporterait pas de garanties réelles. Une idée serait donc de simuler l'exécution d'un code modifié pour identifier son impact sur la machine

Le modèle de faute choisi dans la section 4.2 est un choix basé sur un scénario favorable pour l'attaquant (usage d'un laser, grande précision sur l'emplacement et le moment de l'attaque). La question est maintenant de savoir dans quelle mesure une attaque en faute réussie contre du *bytecode* a un impact sur la sécurité de la carte suivant notre modèle de faute.

Afin de qualifier l'impact, il est nécessaire d'identifier comment la machine se comportera. En l'absence des contre-mesures dédiées dans la section 4.2, la machine virtuelle possède déjà des contre-mesures "naturelles" (c'est-à-dire après chargement et vérification par un potentiel vérificateur de *bytecode*) :

détection de *overflow* ou *underflow* de la pile JCVM : théoriquement, la taille maximale de chaque *frame* est embarquée dans le *bytecode* de chaque méthode. Ainsi, il est possible de détecter une modification du *bytecode* qui entraîne un dépassement de pile, un dépilement sur une pile vide ou la fin d'une méthode finissant avec une pile de taille différente lorsque la méthode a commencé.

opérandes non valides : si certaines opérandes référencent des entrées dans le constant pool, une modification pointant sur un index non-existant est détectable par la machine (il faut noter que cette idée est plus complexe dans la réalité car la machine virtuelle effectue l'édition des liens au chargement afin de remplacer l'opérande par une adresse en mémoire, ceci afin d'éviter une indirection durant l'exécution. Il est donc probable que ce contrôle se fasse plutôt au chargement qu'à l'exécution).

instructions non valides : une instruction *bytecode* a une taille de 8 bits, mais l'ensemble possible de 256 instructions n'est pas utilisé. Il est donc possible de détecter un appel à une instruction non existante.

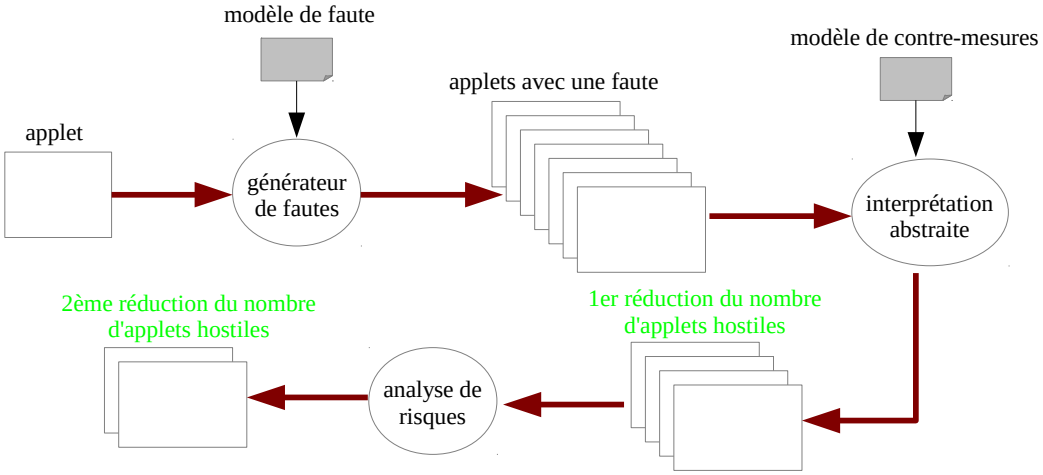


FIGURE 5.1 – Flot d'analyse de SmartCM

L'outil SmartCM, développé par plusieurs étudiants que j'ai encadrés, permet de simuler des attaques sur le *bytecode*, puis d'étudier les impacts et le taux de détection par différentes contre-mesures. Cet outil a été développé pour répondre à trois groupes de questions :

- nous désirions connaître le comportement naturel de la machine virtuelle vis-à-vis des attaques en faute. Par sa sémantique, le *bytecode* n'est il pas naturellement immune contre de telles fautes ? De plus, est-ce que les mécanismes naturels de défense d'une machine virtuelle (pare-feu, contrôle de la pile, etc.) ne limitent pas l'impact d'une faute ?
- il nous fallait évaluer l'efficacité des contre-mesures proposées. Lorsqu'une modification sur le *bytecode* est injectée, dans quel mesure peuvent-elles détecter cette attaque ? De plus, sont-elles suffisamment rapides pour éviter une fenêtre d'opportunité pour un attaquant ? C'est-à-dire ont elles un temps de latence de détection faible (en nombre d'instructions Java Card exécutées) ?
- quelle est la portée réelle d'un applet devenu mutant qui n'est pas été détecté ? Peut-il réellement avoir un impact et ouvrir des nouvelles possibilités pour un attaquant ?

SmartCM est un outil composé de deux composants : un simulateur de faute modifiant un fichier CAP et un interpréteur abstrait capable de vérifier si l'exécution du code modifié est valide. Plus exactement, à partir d'un fichier CAP, d'un modèle de carte et d'un modèle de faute, SmartCM calcule l'ensemble des modifications possibles du *bytecode*, c'est-à-dire l'ensemble des fichiers CAP modifiés par une faute. Le modèle de carte représente ici la manière dont l'impact d'une faute agit sur un élément mémoire, notamment la nouvelle valeur après attaque (soit `0x00` ou `0xFF` selon la technologie mémoire visée, soit une valeur aléatoire si la mémoire est chiffrée).

Pour chacune de ces versions modifiées et pour un profil de contre-mesures, l'outil effectue une exécution abstraite du code afin de déterminer si une exécution du code malicieux est détectée. Le profil de contre-mesures correspond à une ou plusieurs contre-mesures différentes intégrées dans la carte (soit les contre-mesures naturelles, soit les contre-mesures que nous proposons dans la section 4.2). La détection d'une erreur se

déroule comme suit : si, durant l'exécution, la méthode processAPDU est retournée sans erreur ou si une exception est reçue sans être jamais capturée, alors l'outil considère que la faute n'a pas été détectée.

L'étape suivante consiste à qualifier chacun des résultats valides afin de voir quel est leur impact réel. L'outil reconstitue le code Java à partir du *bytecode* modifié par la faute afin que le programmeur puisse comprendre comment la faute arrive à modifier l'exécution. Nous avons donc un outil permettant à un programmeur de comprendre l'ensemble du code mutant produit à partir de son applet. Cet outil a donc une application pratique immédiate et utilisable par des développeurs.

Dans un second temps, ce travail a été étendu afin de s'intégrer dans une réalité industrielle. En effet, lors de la mise en place d'une collaboration avec l'opérateur SFR, l'outil a montré une sérieuse limitation : la première approche a le défaut de générer un nombre important d'applets modifiés mais considérés comme valides du point de vue de l'exécution. Un développeur Java Card voulant connaître l'impact d'une faute sur son applet est dans l'obligation de parcourir un nombre important de résultats et d'interpréter par lui-même la dangerosité de l'attaque.

Pour résoudre ce problème, une seconde phase de développement de SmartCM a abouti à la réalisation d'un analyseur de risque qui regarde si les changements effectués dans le code sont considérés comme sensibles. Nous avons qualifié comme sensibles les cas suivants :

— accès à des méthodes non autorisées ;
— modification du flot d'exécution : désactivation de tests, de throw ou de catch d'une exception ;
— confusion de type d'un paramètre d'une méthode.

5.6 Bilan

L'outil est SmartCM est maintenant opérationnel et atteint ses objectifs répondant aux problématiques que j'ai posées au début de ce document :

— il s'intègre dans l'écosystème Java Card et peut servir à tous les partenaires post-encarteurs ;
— il est efficace pour indiquer aux développeurs d'applets des chemins d'exécutions détournables selon un modèle de faute et les contre-mesures attendues dans la carte ;
— l'information est exploitable par le développeur : afin d'éviter les multiples faux positifs, le module d'analyse de risque réduit le nombre de mutants à vérifier de l'ordre de 90 à 97 % selon l'application.

Par contre, l'analyse statique pour énumérer tous les chemins d'exécutions induits par l'ensemble des fautes possibles sur un applet doit être refaite pour chaque changement (même minime) de code de l'applet. Ce défaut peut entraîner un cercle infernal dans lequel le programmeur engendre de nouveaux chemins pouvant amener à une vulnérabilité lors de corrections des alertes levées par la précédente analyse. Le risque est de voir alors apparaître des structures de données et des méthodes n'ayant aucune relation avec la logique de l'application mais principalement pour servir de barrière contre les impacts de changements par un programmeur. Même si ce défaut est réduit par le module d'analyse de risque, il est néanmoins présent. Il n'a pas été possible de quantifier cet effet de bord possible.

Au lieu de subir ce cycle de correction / impacts, une première piste permettant d'affiner la disposition des contre-mesures serait de générer des contre-mesures avec les résultats produits par SmartCM. Ainsi pourrait-il détecter les portions de code susceptibles d'être affectées par une attaque en faute. À partir de là, il peut soit automatiquement placer des contre-mesures (voire choisir la plus adaptée), soit demander à l'utilisateur d'étudier chaque séquence résultant d'une attaque en faute pour confirmer ou infirmer l'ajout d'une contre-mesure.

Une seconde piste étendant la première est de caractériser les biens à protéger sous forme d'annotations : variables contenant une clé secrète, méthode garantissant l'atomicité d'une opération, etc. Le programmeur peut les identifier explicitement dans son code Java grâce à des mécanismes comme les Xdoclet. Le standard Java Card 3.0 propose même une annotation `@Secure` dont le comportement n'est pas défini et utilisable à la convenance de la machine virtuelle.

Mais au lieu d'indiquer quelle contre-mesure appliquer sur chaque méthode à protéger, le programmeur utilise ces annotations pour indiquer son niveau de sécurité. Par exemple, le programmeur peut marquer une portion de code pour avoir la garantie que l'intégralité de la méthode s'exécute correctement, ou que le contenu d'une variable n'est jamais modifié par une attaque en faute. Cette information pourrait être ensuite appliquée à la pièce logicielle décrite dans le précédent paragraphe. Ainsi, il serait possible de détecter à partir d'une analyse statique si l'attaque en faute peut affecter les propriétés de sécurité des objets et méthodes annotées. Il ne faut pas oublier qu'il reste au programmeur la charge de décider quel élément doit être sécurisé et avec quel niveau de sécurité, et il semble difficile de pouvoir concevoir cette étape de manière entièrement automatisée sans intervention humaine.

5.7 Ressources

Participants

Mes travaux sur l'analyse de conformité des cardlets se sont appuyés sur les développements réalisés par Yorick Lesecque et Thibault Desmoulins autour de FIndBugs. J'ai accompagné par un support scientifique et technique la transmission industrielle à la *spin-off* Arya Security menée par Clément Mazin et Jean-Baptiste Machemie [CIC4].

J'ai co-encadré les travaux sur la sécurité des applications web dans les Java Card réalisés par Nassima Kamel [ST2], doctorante entre 2009 et 2012. Ces travaux ont été financés par le projet FUI MECANOS, regroupant des acteurs industriels comme Orange, SFR ou Gemalto.

Concernant l'outil SmartCM [CI1], la première version de cet outil a été réalisée par Ahmadou Séré durant son travail de doctorat [ST4]. J'ai ensuite encadré les stagiaires Jean-Baptiste Machemie et Lonny Brissac pendant la reprise de ce travail afin d'améliorer les résultats, puis les étudiants de Master 1 Gael Le Dantec, Patrick Silvera, Jeremy Fridman et William Betremieux durant le développement du module d'analyse de risque durant leur projet de Master 1 et enfin les ingénieurs Clément Mazin et Jean-Baptiste Machemie pendant le développement de l'ensemble des contre-mesures.

Outils

Analyse statique d'applet web

Dans un premier temps, nous avons développé pour Java Card 3.0 un applet de disque virtuel [CNC3] permettant de proposer à un utilisateur de stocker ses fichiers personnels dans différents serveurs distants en chiffrant par l'intermédiaire de l'application embarquée dans la carte les données. La création de ce logiciel nous a permis de comprendre les problématiques associées au développement d'une application web dans une telle carte. Nous avons alors écrit à destination de nos partenaires du projet MECANOS un guide de bonnes pratiques contenant un ensemble de recommandations pour sécuriser son application [CI7].

Les travaux d'analyse statique d'applet web ont été diffusés à nos partenaires du projet MECANOS. Le portage de la librairie JC3XSSFilter [6] est maintenant en open-source.

Simuler les fautes

L'outil SMartCM, comprenant l'outil d'injection de faute dans un fichier CAP, l'outil d'exécution abstraite de l'application contenue dans ce fichier et l'outil d'analyse de risques, a été transféré à l'industrie à notre partenaire SFR.

6. https://bitbucket.org/ssd/jc3xssfilter

Chapitre 6

Autre Travaux

Plusieurs autres de mes travaux dans l'équipe SSD n'ont pas été référencés dans ce document car extérieurs au domaine de la Java Card. J'ai donc choisi de les présenter indépendamment.

6.1 Authentification par test de Turing graphique sur mobile

Les travaux présentent un mécanisme de transaction commerciale pour mobiles (m-transactions) équipés d'une carte à puce [CIC5, CN1]. Le dispositif est conçu pour resister à la capture graphique et à l'utilisation des autorisations par un malware embarqué dans le téléphone. L'idée principale est la génération d'un CAPTCHA [1] composé du montant de la transaction et d'un code secret généré pour chaque transaction. Le symbole graphique est connu pour être généré par la carte à puce et être suffisamment offusqué pour empêcher toute analyse par reconnaissance de la part de logiciel malveillant. Cette approche permet de répondre au manque de confiance dans un portable en déportant l'autorisation avec le service distant dans la carte à puce.

6.2 VoIP complètement anonyme

Nous avons présenté une méthode pour dissimuler le trafic de communications VoIP [CIC7]. Même si des mécanismes de chiffrements cryptographiques existent pour la VoIP dans un contexte point-à-point, il est difficile d'achever un complet anonymat, car des entités avec des capacités d'écoute sur le réseau peuvent déduire des métadonnées comme l'émetteur, le destinataire, la durée de l'échange, les caractéristiques des flux à partir des échanges de négociation, etc. La solution utilise le principe cryptographique PIR (*Private Information Retrieval*) en maintenant des flux entre tous les clients même en l'absence de communication. En l'absence de connaissance des clés de chiffrement, un attaquant ne peut distinguer du bruit des véritables échanges en cours dans l'ensemble du trafic (y compris l'identité des clients dont une communication est en cours).

1. Completly Automated Public Turing test to tell COmputers and Humans Apart

6.3 Réseau domestique sécurisé

Nous avons présenté un protocole pour la distribution de clé dans des appareils domestiques afin de sécuriser la communication entre objets de manière simple [CI2]. En s'appuyant sur la disponibilité de matériel de confiance emportant une pile Java Card, le protocole permet à l'utilisateur de sécuriser à l'aide d'un code simple les communications sans jamais connaître la clé utilisée lors du chiffrement. Cette approche permet d'offrir des canaux sécurisés contre des attaquants, qu'ils soient externes ou utilisateurs du réseau, ce qui permet le transport de DRM de manière simplifiée. Néanmoins, l'utilisateur peut révoquer les objets qu'ils ne désirent pas accéder dans son réseau.

6.4 Sécurité des mécanismes d'hypervision

Des travaux se sont intéressés à évaluer la sécurité des mécanismes de virtualisation pour les systèmes embarqués dans deux domaines d'activité. Dans le premier je me suis intéressé à l'hypervision pour la téléphonie mobile et j'ai proposé avec Amaury Gauthier et Jordan Bouyat des mécanismes de fuzzing des "hypercalls" [WI1, CN3, CI4]. Le second domaine concerne les mécanismes de virtualisation dans les systèmes temps-réel. Avec Pierrick Buret, nous sommes en train de proposer des méthodes d'évaluation du WCET à partir d'algorithmes génétiques considérant non pas les entrées du code à évaluer mais aussi son contexte d'exécution (*i.e.* les états de l'hyperviseur) [CIC1].

Chapitre 7

Bilan

L'étude principale abordée dans ce document concerne la sécurité des cartes à puces Java Card. Elle s'intéresse plus particulièrement aux applications (applets) comme **cible d'attaque** mais aussi comme **vecteur d'attaque**.

Les applications sont en effet des cibles intéressantes par l'excès de confiance dans l'isolation offerte par la machine virtuelle Java Card. Premièrement, j'ai démontré avec mon équipe que l'usage de méthode d'évaluation rapide ("fuzzing") est possible pour évaluer l'implémentation de protocoles de communication en tant qu'application Java Card. Deuxièmement, j'ai démontré par exécution abstraite les menaces des attaques en fautes sur le *bytecode* des applications et les impacts possibles, notamment par les possibilités d'outrepasser des contrôles effectués dans l'application.

Si l'attaquant est en capacité de charger (ou de faire charger) des applications forgées ou modifiées par ses soins, les charges applicatives deviennent alors des vecteurs d'attaques contre la machine virtuelle. Ainsi, mes travaux ont montré l'existence d'une faiblesse dans les spécifications Java Card permettant d'outrepasser le contrôle du pare-feu et d'accéder à différents plans mémoires de la carte, de modifier le code d'autres applications Java Card jusqu'à exécuter du code natif en dehors de la machine virtuelle. De plus, j'ai développé avec mon équipe la notion d'applet mutant (des applications conformes du point de vue du vérificateur de *bytecode* mais forgées pour outrepasser les protections de la machine virtuelle), concept depuis validé par plusieurs autres travaux scientifiques.

Après avoir rempli mes objectifs en terme de nouvelles approches d'attaques contre les cartes, mes travaux se sont intéressés à la nécessité de contre-mesures pour lutter contre ces menaces. Je présente alors une gamme de solutions qui s'appuie sur l'idée que la machine virtuelle (telle que définie actuellement par les spécifications Java Card) n'est pas suffisante pour lutter contre les attaques en fautes ou les faiblesses de développement d'applications Java Card. Je présente alors des propositions, outils et méthodologies afin de renforcer la machine virtuelle mais aussi les charges logicielles :

- En étendant l'outil d'exécution abstraite pour identifier les potentielles attaques en faute sur un applet, j'ai proposé une méthode pour réduire les faux positifs et réduire ainsi le code à renforcer par un développeur ;
- A l'aide d'un outil d'analyse statique, j'ai proposé un outil pour renforcer la qualité de développement d'applet en détectant des motifs de code déconseillés par des référentiels de bonnes pratiques ;
- Par la génération de méta-données hors-ligne associées au code de l'application à charger afin de permettre à une machine virtuelle Java Card modifiée de vérifier que l'exécution de l'applet est l'une de celles attendues ;

— Par l'ajout d'une mécanisme de mise à jour dynamique dans une machine virtuelle Java Card afin de réduire la taille des transferts et de ne pas perdre le contexte d'exécution, offrant alors plus de souplesse pour gérer les mises à jour de sécurité des applications.

7.1 Originalité de l'approche

Je considère que mon approche menée et développée dans l'équipe SSD est originale pour plusieurs raisons.

Premièrement nous avons choisi d'aborder les aspects offensifs et défensifs afin de développer une expérience pratique de la sécurité des cartes. Cette approche nous a permis d'acquérir des compétences spécifiques sur les deux aspects.

Deuxièmement, elle montre l'intérêt qu'offre l'étude de la sécurité des applications au lieu de se limiter à la sécurité des machines virtuelles, des algorithmes cryptographiques et du support physique selon deux vecteurs : le fait que ces applications puissent être corrompues et amener à la divulgation de leurs secrets, mais aussi le fait que les applications peuvent être des vecteurs d'attaques sur le système sous-jacent.

Troisièmement, nous avons réussi à montrer plusieurs faiblesses dans l'implémentation de protocoles de communication mais aussi dans la spécification même de Java Card. Certaines de ces attaques ont notamment permis la découverte de plans mémoires complets de cartes Java Card disponibles sur le marché.

Quatrièmement, nous avons combiné plusieurs approches pratiques d'évaluation de la sécurité, en proposant l'usage de l'analyse statique pour la sécurité des applications, du fuzzing sur les protocoles de communication, de la recherche de vulnérabilité dans les spécifications jusqu'à l'exploitation des mécanismes internes d'instance de Java Card.

Cinquièmement, nous avons proposé plusieurs solutions à différents niveaux du cycle de développement (définition, conception, évolution) qui sont non seulement adaptés au flot de développement Java Card et aux contraintes industrielles, mais aussi bénéfiques pour l'écosystème Java Card (par la disponibilité de nos travaux et outils). Cette volonté de transfert vers l'industrie s'est consolidée par la création de la spin-off Arya Security.

Ainsi, ces approches et stratégies sont récompensées par les résultats suivants :

— Aucune équipe académique de recherche ne travaillait en 2008 sur la sécurisation du *bytecode* contre les attaques en faute. Nous avons été novateurs en montrant l'existence de cette menace et en démontrant les possibilités d'exploitation, notamment avec le simulateur SmartCM et le concept d'applets mutants. Nous avons ainsi permis la prise de conscience de ce problème à tous les acteurs (encarteurs et développeurs d'applets), ce qui a débouché sur le projet Inossem regroupant l'ensemble de ces acteurs afin de proposer une standardisation des mécanismes de sécurité contre ces attaques.

— Peu d'équipe de recherche ont proposé des résultats pratiques d'exploitation de cartes à puce Java Card (à part les laboratoires d'Oberthur et les travaux de Poll) et obtenu l'accès aux mécanismes internes de fonctionnement de la carte. Signe de cette reconnaissance, les attaques EMAN et la classe d'attaques des applets mutants sont aujourd'hui évaluées par les CESTI (Centres d'Évaluation de la Sécurité des Technologies de l'Information) lors de la certification de Java Card et de leurs applications.

— Plusieurs outils développés dans notre équipe ont obtenu une certaine reconnaissance industrielle :
  — OPAL est aujourd'hui l'implémentation de référence de la norme GlobalPlatform ;
  — La version restreinte de CapMap a été diffusé à plusieurs industriels du secteur.

7.2 Bilan personnel

Du développement de projet scientifique

D'un point de vue personnel, l'expérience de recherche sur Java Card a été une des plus enrichissantes. Le fait de pouvoir travailler avec le professeur Jean-Louis Lanet a amené son expérience pratique de la sécurité industrielle avec des objectifs permettant non seulement de trouver des nouvelles approches et solutions scientifiques, mais aussi de les considérer vis-à-vis de la réalité des solutions industrielles et de leurs intégrations.

Une des conséquences immédiates est la nécessité de trouver de nouveaux vecteurs d'attaques dépendant d'implémentations de Java Card et par l'absence d'une litterature scientifique importante, il faut développer une stratégie d'expérimentation en interne afin de les découvrir.

Les relations industrielles sont donc un point majeur de nos recherches. Il est nécessaire de pouvoir construire une relation de confiance et d'intégrer l'écosystème existant pour prendre connaissances des contraintes qui s'appliquent au développement de nos solutions (métriques d'évaluation par exemple). Développer des outils et des solutions pertinentes, évolutives et réutilisables est une des meilleures méthodes qui soit pour arriver à cet objectif.

Cette connaissance de l'écosystème est aussi nécessaire dans le cadre de montage de projets car les objectifs peuvent être amenés à modifier l'équilibre de ces projets (notamment par de nouveaux modèles commerciaux). Cette précaution est d'autant plus importante que la recherche de partenariats dans le cadre de financements à débouchés industriels implique des membres de cette communauté.

De l'enseignement par la recherche

Dans notre activité, un des points clés découvert au fur et à mesure de notre recherche est de trouver des étudiants motivés pour participer à nos travaux par le développement d'outils et/ou pour travailler sur des problématiques de recherche.

Commencer à intéresser un étudiant lors d'un projet de recherche dans le second semestre de Master 2 est un peu tardif car il n'a pu construire une expérience préalable (en dehors de la formation dispensée).

La stratégie est d'identifier et d'intéresser des étudiants motivés et curieux afin de les accueillir dans l'équipe au plus tôt (certains de nos travaux qui ont menés à publication ont été réalisés par des étudiants de Licence 3 et Master 1) :

— identifier des étudiants prometteurs dès la Licence 1 et trouver des opportunités (projets ou stages d'été) pour les intégrer à l'équipe ;
— développer des cours au plus tôt dans la formation pour augmenter le niveau sur des sujets connexes, dans notre cas les systèmes embarqués par exemple ;

— mettre en place une politique de sujets ambitieux (de la part d'industriels ou d'anciens de la formation) afin d'avoir des profils travaillant sur des technologies et avancées pertinentes et à jour.

À ce propos, la sécurité est un excellent vecteur de formation : un étudiant ne peut avancer dans une approche défensive ou offensive sans connaître le fonctionnement et les interfaces accessibles de l'entité en train d'être évaluée. La capacité d'auto-apprentissage et de remise en question nécessaires dans ce cas sont deux qualités indéniables d'un candidat à une thèse.

Un autre point est le développement d'outil par les étudiants. Si l'objectif est d'obtenir des outils diffusables et "professionnels", alors il est nécessaire de renforcer la qualité de développement :

— développer avec des outils de gestion de sources (git) ;
— former les étudiants sur le test et l'intégration continue (souvent absent des formations universitaires) ;
— avoir une boucle de retour courte : forcer le développement de petits prototypes régulier afin de valider l'avancement.
— accepter dès le début l'échec possible du projet.

Chapitre 8

Perspectives

"And now for something completely different..."

— John Cleese, *Monty Python Flying Circus*

Depuis septembre 2013, je suis Maître de Conférences à l'Université de Lille où j'ai intégré l'équipe 2XS spécialisée sur les petits objets embarqués sécurisés ("eXtra Small eXtra Secure"). La cible est plus importante que mes précédents travaux, mais néanmoins de taille limitée : principalement les processeurs embarqués. Dans cet optique, j'ai intégré avec mes compétences en sécurité cette équipe pluridisciplinaire dans les aspects économie d'énergie et méthode formelles.

8.1 Contexte

La sécurité d'un système dépend de la confiance dans le logiciel et le matériel, mais aussi du modèle d'attaquant considéré. L'hypothétique modèle d'un attaquant possédant des ressources illimitées est aujourd'hui considéré comme réel, notamment par les suites des révélations sur les programmes d'espionnage de certains pays mais aussi l'écosystème de la criminalité informatique basé sur des stratégies de "drive-by-download" [46] ou encore les exploitations massives dans les systèmes embarqués de type IoT (*Internet of Things*) ou SCADA (*Supervisory Control And Data Acquisition*) connectés à Internet [67, 5].

Apporter la confiance dans de tels systèmes est difficile car les outils de développement sont construits avec des objectifs de performance, la sécurité étant reléguée à un second plan (dans le meilleur des cas). Les langages de développement pour les systèmes embarqués (C, C++, assembleur) ne sont pas construits pour éviter les détournements d'exécution [107] et ne sont pas exempts eux-mêmes de certaines faiblesses [17]. Le renforcement de la sécurité par des outils d'analyse statique ou dynamique du code, ou la génération de tests n'apporte pas de réelles certitudes, la complexité de l'analyse limitant son efficacité [13] et les tests ne garantissant pas l'absence de bugs [113].

Concernant les supports matériels d'exécution, la présence de dispositifs de protections (comme la non-exécution de la pile d'exécution ou la randomisation de l'adresse de base en mémoire des éléments d'un exécutable) n'offrent pas de garanties suffisantes [33]. Mais surtout, la complexité du matériel et l'usage de micro-codes entraînent de nombreuses failles au niveau des périphériques [26] et dans le CPU de la plate-forme [45].

Du développement à l'exécution, la complexité croissante et le manque de consistance des piles logicielles et matérielles ne permettent pas d'établir le développement de systèmes sécurisés. L'approche utilisée couramment est la présence de contre-mesures locales à des attaques spécifiques. Par exemple, des attaques par le matériel exploitant l'absence

de contrôle lors de l'utilisation du DMA [30] sont aujourd'hui compensées par l'IOMMU, un dispositif de protection des accès mémoires de la part des périphériques [40].

Une autre approche est de renforcer l'isolation entre composants de différents niveaux de criticité, avec l'usage au niveau logiciel de mécanismes de virtualisation / isolation à différents niveau de la pile logicielle du système (conteneurs [58], para-virtualisation [82], *full virtualisation* [61]). Au niveau matériel, les mécanismes de virtualisation comme Pacifica/AMD-v [69] et Intel VT-x [75] offrent des mécanismes facilitant l'exécution de plusieurs systèmes d'exploitation au-dessus d'un hyperviseur. Une solution alternative sur l'architecture ARM est TrustZone [43], un environnement d'exécution privilégié parallèle au niveau noyau et utilisateur afin d'exécuter du code critique de confiance indépendamment du système d'exploitation potentiellement non sécurisé.

Mais il n'existe pas de confiance dans le matériel ou le logiciel, l'approche actuelle consiste à d'une part réduire le risque et d'autre part une politique de mises à jour en cas de de l'existence d'une faille.
La confiance dans la carte à puce est passée de l'obscurité de ses composants vers une confiance ouverte, grâce à la présence de certification de haut niveau impliquant des méthodes formelles pour garantir la confiance des composants logiciels et matériels. La carte à puce a ouvert la voie à un processus qu'il faut maintenant porter dans les systèmes embarqués.

8.2 Propositions

Proposition 1 : prouver le matériel

Par l'obscurité de ces composants et par les nombreuses possibilités d'introduction de fonctions malveillantes, les solutions matérielles, quelque soit leur complexité, ne possèdent aucune garantie autre que celle offerte par le fondeur. Le logiciel appuyant ses propriétés de sécurité sur un comportement attendu du matériel, il est nécessaire de **prouver le matériel**. Comme il n'est pas possible aujourd'hui de prouver du matériel existant ou futur par la difficulté de construire une preuve forte à partir du silicium, la preuve doit être **construite à partir d'un modèle formel du matériel** et garantie lors de la fabrication du composant (probablement de manière contractuelle).

Si une confiance existe au niveau du matériel, une preuve peut alors être construite sur le logiciel. Une stratégie est de réduire la taille du code s'exécutant au niveau privilégié du processeur, identifié alors comme la base minimale de confiance (TCB ou *Trusted Computing Base*). De nombreux travaux sur les micro-noyaux comme Minix [65] ou plus récemment OKL4 [39] ont démontré la faisabilité d'un micro-noyau ne supportant que l'isolation mémoire, le mécanisme d'échange de messages (IPC) et l'ordonnanceur. SeL4 [8] est l'un des projets les plus aboutis en terme de sécurité, avec le développement du modèle Isabelle d'un micro-noyau L4 [1], d'un ensemble de preuves sur ce modèle et d'une implémentation vérifiée de bout-en-bout (*i.e.* le binaire du micro-noyau est cohérent vis-à-vis de la spécification).

1. relhttp ://www.l4hq.org/

Par la démonstration d'un micro-noyau complètement prouvé, SeL4 possède une preuve importante (plus de 100.000 lignes de code Isabelle représentant un travail de 25 personnes-années). Mais la taille de cette preuve entraîne une rigidité du modèle : l'ajout, le retrait ou la modification de parties du modèle entraîne un travail potentiellement important de reconstruction des preuves. Les développeurs avec une compétence limitée dans les preuves ne peuvent donc pas adapter le micro-noyau pour des usages particuliers.

Or il existe certaines restrictions dans le modèle afin de limiter la taille et la faisabilité de la preuve. Premièrement, le mécanisme d'isolation mémoire ne protège que l'isolation du noyau, les pages physiques sont mises à disposition des partitions dans l'espace utilisateur par un système de capacité (*capabilities*) ne garantissant pas l'isolation entre processus [2]. Le développeur doit donc utiliser des stratégies comme une décomposition physique statique de la mémoire, développer le code de chaque partition en distribuant les responsabilités ou encore le développement d'un gestionnaire de mémoire virtuel prouvé pour apporter certaines garanties. Deuxièmement, l'ordonnanceur est de type *round-robin* préemptif avec 256 niveaux de priorités qui n'est pas adapté à certains contextes d'usage particulier nécessitant des contraintes temps-réel par exemple.

Afin de réduire le travail sur la preuve, une première approche peut se concentrer sur l'outillage. SeL4 a utilisé des piles logicielles de solveur de preuves indépendantes du flot de développement du système. Le travail de Prove&Run, industriel travaillant avec notre équipe est de développer actuellement cette pile intégrant les deux aspects, réduisant alors le temps de développement de systèmes embarqués de confiance.

8.3 MesoVisor

Proposition 2 : réduire à l'extrême le TCB

La nécessité d'avoir de la flexibilité au niveau du TCB est en contradiction avec l'aspect rigide de la preuve. Je propose de réduire à l'extrême la taille du code privilégié en assurant uniquement l'isolation mémoire. Cette fine couche d'exécution baptisé "MesoVisor" n'aurait comme but d'assurer l'isolation mémoire du code s'exécutant en espace utilisateur mais aussi en espace noyau.

Le fondement de cette approche s'appuie sur l'idée qu'une fois l'isolation mémoire établie et prouvée, les propriétés de sécurité sont plus faciles à identifier car les dépendances entre composants isolés sont réduites. Cette approche permettrait la composition de certification ("Lego©-certification") dans les systèmes embarqués, dont la sécurité est actuellement construite sur un design simple d'architecture, des référentiels de sécurité, le test éventuellement complété par l'analyse statique.

L'isolation est vue ici comme une propriété forte, indépendante du mode de privilège du système. Le code s'exécutant en mode utilisateur serait contenu de manière classique dans les pages de mémoire physique attribuées par la MMU. Mais pour des raisons de performance, il est difficile d'envisager que des fonctionnalités comme l'ordonnanceur ou IPC soient exécutés au niveau de privilège utilisateur.

2. Cette approche semble plus une *feature* qu'un réel bug : les concepteurs de SeL4 voulant probablement offrir un modèle de gestion mémoire flexible. Néanmoins, cela impose d'avoir un ensemble prouvé de la gestion mémoire des processus pour apporter la confiance.

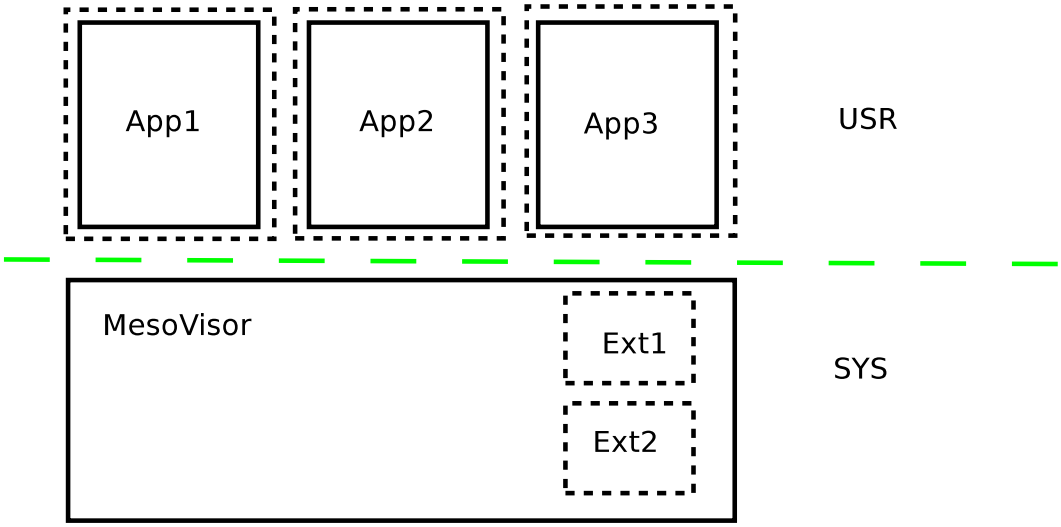


FIGURE 8.1 – Modèle du MesoVisor

Proposition 3 : isolation du code au niveau noyau

Des travaux récents étudiés dans notre équipe ont montré la possibilité d'isoler du code au niveau du noyau en réécrivant celui-ci avant exécution. Je propose un mécanisme permettant l'ajout de code au niveau noyau avec des propriétés fortes garantissant son isolation, c'est-à-dire que les branchements d'exécution ne peuvent échapper à l'isolation et que l'accès à la MMU n'est pas possible.

Cette approche peut paraître peu performante mais nous sommes dans une optique d'isolation, donc si le code isolé dans le noyau est écrit pour être performant dans cet environnement le surcoût sera limité. Mais surtout, ce code apportera des fonctionnalités performantes dans le noyau sans remettre en question la preuve sur l'isolation mémoire.

8.4 Programme

De part la construction d'un système prouvé au niveau logiciel et matériel s'ouvre la possibilité de définir un modèle d'isolation abstrait de l'implémentation logicielle et/ou matérielle. Ainsi, le modèle proposerait une preuve d'isolation sans à se préoccuper des restrictions matérielles. Deux implémentations du "MesoVisor" seraient possibles :

— une version s'appuyant sur une couche matérielle prouvée ;
— une version s'appuyant sur une MMU traditionnelle.

L'objectif n'est pas de définir une preuve de bout-en-bout pour plusieurs raisons. Premièrement, le but est de valider l'idée mais une vérification complète est en dehors de notre portée. Dans ce cadre, notre collaboration avec Prove&Run permettrait un transfert technologique car ses équipes sont tout à fait à même de mener ce développement intégral. Deuxièmement, seL4 a déjà démontré la faisabilité de prouver un noyau de bout-en-bout, le fait de développer le noyau complètement n'apportera rien en terme scientifique.

Dans un premier temps, le modèle Haskell de l'architecture ARM disponible dans le projet SeL4 sera utilisé afin de pouvoir construire la collaboration entre les personnes travaillant sur la formalisation et celles travaillant sur l'implémentation. L'objectif est

d'avoir le cercle vertueux posé par SeL4 : la capacité que chaque partie puisse influencer l'autre :

— la partie formalisation peut identifier les difficultés de modélisation et les risques d'explosion combinatoire de construction des preuves influencé par le design du noyau et ses optimisations ;
— la partie système peut influencer le modèle d'optimisation et valider le modèle formel.

8.5 Conclusion

SeL4 a montré que réaliser un noyau prouvé de bout-en-bout est possible. Mon objectif n'est pas de refaire leur travail mais de dépasser l'aspect inflexible de la preuve attachée au noyau. J'ai baptisé cette orientation **la seconde génération de noyaux prouvés**, par opposition à SeL4 que je considère comme la première génération. Ce sujet ambitieux est à la portée de notre équipe depuis l'arrivée de David Nowak, spécialisé dans la preuve formelle. L'expérience de 2XS sur les systèmes embarqués notamment en terme de performance est idéal pour atteindre l'objectif d'un tel projet.

Chapitre 9

Ressources

> *"A Thaum is the basic unit of magical strength. It has been universally established as the amount of magic needed to create one small white pigeon or three normal sized billiard balls."*
>
> — Terry Pratchett, *The Light Fantastic*

Les tableaux 9.2 et 9.3 présentent une vision globale de l'affectation des ressources (humaines et projets) à chaque axe et thème.

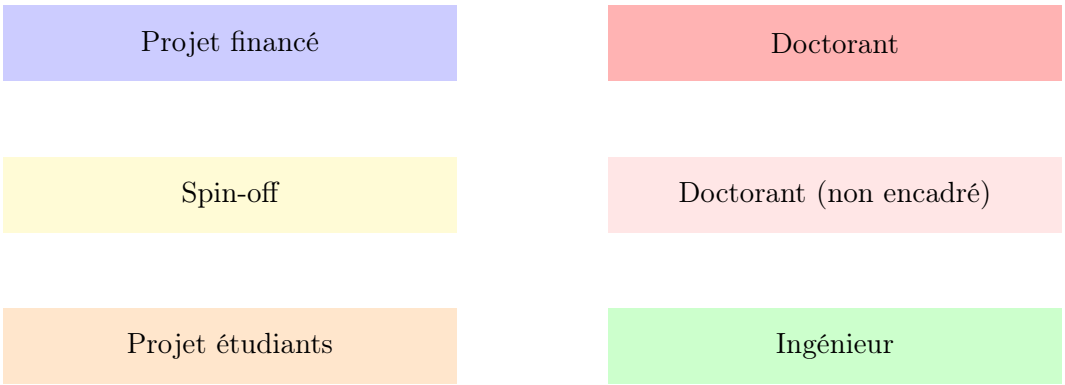


FIGURE 9.1 – Légende des ressources

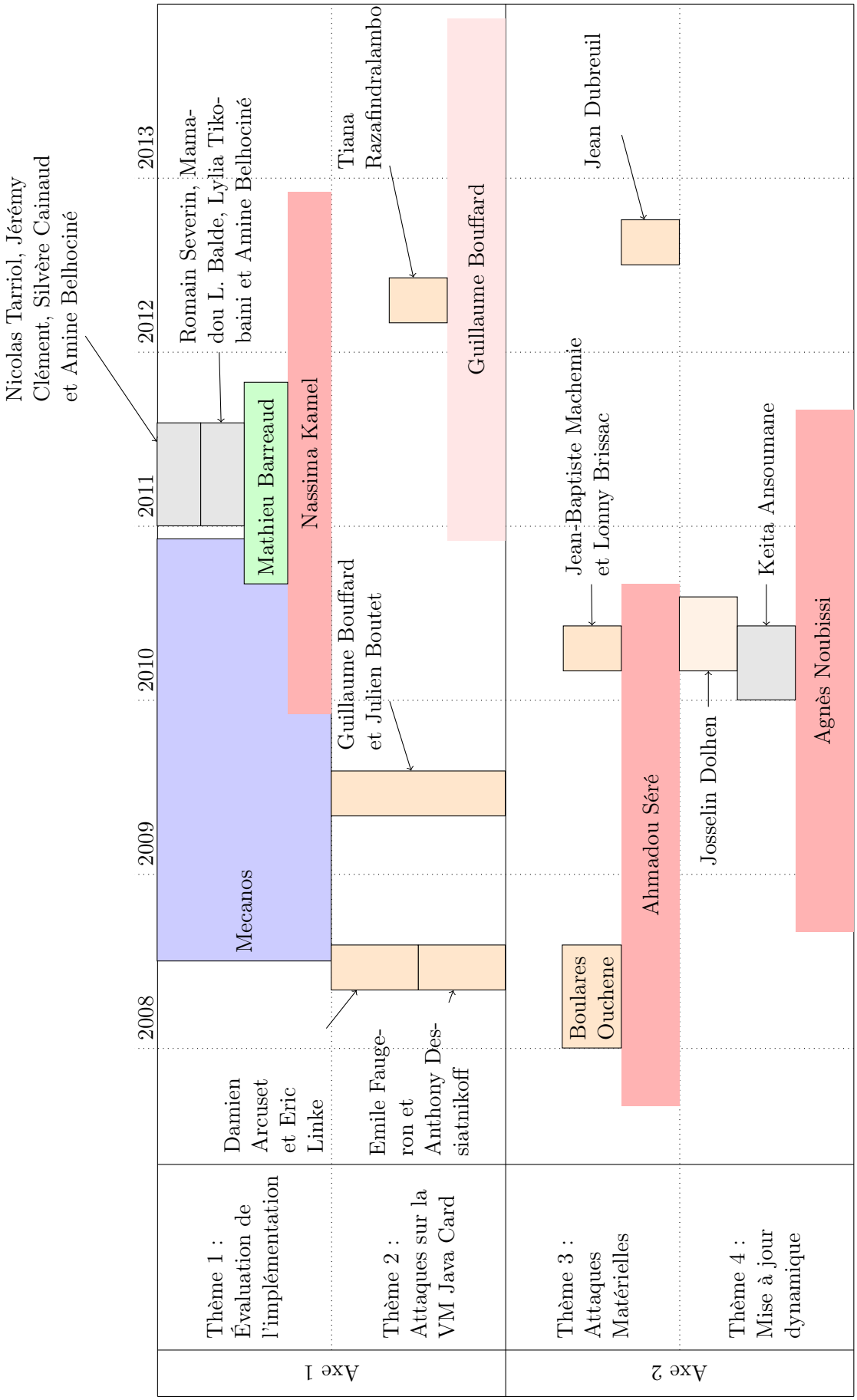


FIGURE 9.2 – Détails des ressources de l'axe 1 et 2

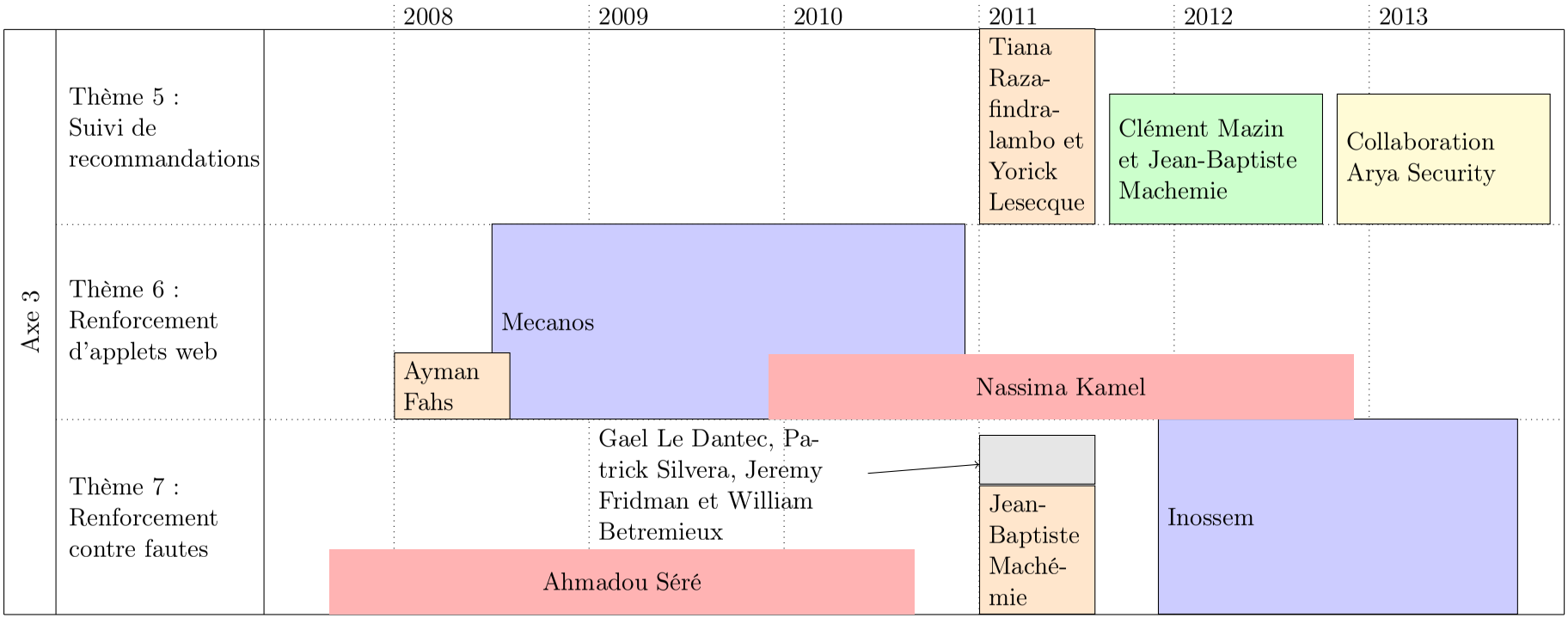


FIGURE 9.3 – Détails des ressources de l'axe 3

9.1 Projets

Méthodologie pour les Cadres Applicatifs des Nouveaux Objets Sécurisés (Mecanos)

— Juin 2008 – Novembre 2010
— **Financement** : Fonds unique interministériel (FUI), pôles de compétitivité System@TIC et Elopsys
— **Partenaires** : Trusted Labs, Trusted Logic, Gemalto, Oberthur Card Systems, Soliatis, IdConcept et EURECOM
— **Description** : L'objectif est d'accélérer l'adoption par le marché des nouvelles générations de carte à puce (Java Card 3) en apportant une réponse concrète aux verrous identifiés (absence de plate-forme adaptée, manque d'outils, de puissance de calcul, de capacité de stockage, protection contre les attaques logiques perpétrées à distance,...) et en s'intéressant à tous les points indispensables au déploiement d'une telle technologie (méthodologie de développement et de sécurisation pour les développeurs et pour les organismes de certification, méthodologie de tests fonctionnels et de tests sécuritaires).
— **Participation** : Dans le cadre de la thèse de Nassima Kamel [ST2], nous avons proposé un guide méthodologique de bonnes pratiques pour le développement d'applications webs à destination d'une carte à puce, une méthode d'évaluation de la résistance de la pile HTTP et un outil d'analyse *offline* des applets webs afin de les prémunir contre les attaques de type *cross-site scripting*.
— **Publications** : [CNC3, CI7, CI5]

Technologie et Investigations Sécuritaires pour téléPHones et Appareils Numériques Mobiles (Tisphanie)

— Sept. 2009 – Oct. 2011
— Financement : Fonds unique interministériel (FUI), pôles de compétitivité System@TIC et Elopsys
— Partenaires : Bertin Technologies, EADS, Trusted Logic, Gemalto, Trusted Labs, CEA-Leti, École nationale des Mines de Saint-Etienne et l'Université Versailles Saint-Quentin
— **Description :** Le but principal du projet TISPHANIE est de proposer une méthodologie structurée et la plus systématique possible, les outils et processus d'évaluation sécuritaires permettant de pouvoir donner à priori aux utilisateurs concernés (opérateurs mobiles, développeurs d'applications, laboratoires de la police,..) une évaluation rapide de la sécurité des « composants clefs » des téléphones mobiles, PDA, terminaux PMR, etc., utilisés pour des applications critiques.
— **Participation :** Nous avons proposé des méthodes d'évaluation des hyperviseurs embarqués dans les téléphones portables.
— **Publications :** [CI4, CN3, WI1]

Sécurité des Systèmes Embarqués de l'Aérospatiale

SOBAS
— Octobre 2010 – Mars 2014

— Financement : Projet ANR, pôles de compétitivité Aerospace Valley et Elopsys
— Partenaires : Airbus, ANSSI, ASTRIUM, EADS-IW et le LAAS
— **Description :** Le but de ce projet est d'étudier la sécurité des nouveaux mécanismes d'hypervision pour les systèmes embarqués critiques utilisés dans l'avionique et l'aérospatiale. Nous nous intéressons particulièrement à la mise en relation entre les propriétés de sécurité désirées, et les détails d'implémentation où se situent la plupart des vulnérabilités : fonctions des noyaux d'OS dédiées à la protection des espaces d'adressage, à la gestion des interruptions et au changement de contextes, etc. ; implémentation matérielle des mécanismes de protections et d'autres fonctions ancillaires, susceptibles d'être exploitées par des attaquants.
— **Participation :** Avec le doctorant Pierrick Buret [ST1], nous sommes en train d'évaluer la résistance de l'hyperviseur XtratuM contre des attaques exploitant des fonctionnalités matérielles non contrôlées par l'hyperviseur.

INter Opérabilité Sécuritaires des Systèmes EMbarqués (INOSSEM)

— Décembre 2011 – Septembre 2013
— Financement : Grand emprunt (technologie de sécurité et résilience des réseaux)
— Partenaires : Trusted Labs, Oberthur Technologies, Bouygues Telecom, SFR, SERMA technologies, GREYC, RATP, SAGEM Sécurité, Orange et Gemalto
— **Description :** Le but de ce projet est de permettre l'intéropérabilité sécurité entre les fabriquants de cartes à puce, notamment vis-à-vis de nouvelles menaces comme les attaques en faute. La principale contribution attendue est la proposition d'une API commune à toutes les cartes à puce, offrant des mécanismes de sécurité de contrôle de l'intégrité du code afin d'aider le programmeur d'un applet à renforcer les contrôles outrepassables par une attaque physique.
— **Participation :** Nous avons proposé une maquette d'une machine virtuelle implémentant l'API inossem afin de valider l'approche proposée dans ce projet.

9.2 Spin-off

Arya Security

— Incubé dans notre équipe depuis Septembre 2011 – Autonome depuis Septembre 2012
— Financement : Détection Inovation Laboratoire
— **Description :** Développement d'un moteur d'analyse statique d'applets pour technologie NFC permettant la certification semi-automatique.
— **Participation :** J'ai encadré la construction de cette équipe en transférant notre savoir sur l'analyse statique d'applet Java et en supervisant la gestion du projet des développement des outils.

9.3 Thèses

Pierrick Buret

— **Titre** : "Étude de la sécurité de l'ordonnanceur de l'hyperviseur XtratuM"
— Débutée en Janvier 2013 – fin prévue en Décembre 2015
— **Financement** : Projet ANR SOBAS (50%) et DGA (50%)
— **Encadrement** : 50% (co-encadré avec le professeur Gilles Grimaud)
— **Sujet** : L'hyperviseur XtratuM permet l'exécution de plusieurs partitions logicielles tout en garantissant l'isolation spatiale et temporelle. Conçu pour des systèmes embarqués dans les satellites, l'ordonnanceur de XtratuM se doit de respecter les aspects temps-réel de l'ensemble des partitions, tout en offrant de bonnes performances et en garantissant la sécurité du système. Nous avons montré que ce n'était pas le cas par l'exploitation possible de caractéristiques matérielles de la plate-forme d'exécution. Notre travail est de proposer un nouvel ordonnanceur respectant les contraintes de l'environnement tout en offrant une meilleure flexibilité.

Nassima Kamel

— **Titre** : "Sécurité des cartes à puce à serveur Web embarqué"
— Débutée en Décembre 2009 – soutenue le 20 Décembre 2012
— **Financement** : Projet FUI MECANOS
— **Encadrement** : 50% (co-encadré avec le professeur Jean-Louis Lanet)
— **Devenir** : Nassima Kamel est ATER à l'Université de Limoges
— **Sujet** : L'ajout du protocole HTTP et d'un serveur web dans une carte à puce peut remettre en cause sa sécurité. Nous proposons dans ce travail un mécanisme d'analyse *offline* permettant de s'assurer que les cartes ne peuvent servir de relais à une attaque *cross-site scripting* (XSS). De plus, nous proposons une méthodologie d'évaluation de la pile HTTP basée sur une approche *fuzzing* renforcée par une analyse des réponses renvoyées et la construction d'une grammaire réduite de HTTP en fonction des fonctionnalités supportées par le serveur web de la carte.
— **Publications :** [CNC3, CI7, CI5]

Agnès Cristèle Noubissi

— **Titre** : "Sécurité dynamique dans le contexte des cartes à puce"
— Débutée en Octobre 2008 – soutenue le 12 Septembre 2011
— **Financement** : Bourse de la région Limousine
— **Encadrement** : 50% (co-encadré avec le professeur Jean-Louis Lanet)
— **Devenir** : Ingénieur R&D à AByster depuis Janvier 2012
— **Sujet** : Nous proposons l'architecture EmbedDSU pour permettre la mise à jour dynamique (*Dynamic Software Update* ou DSU) d'un applet dans une carte à puce Java Card. Cette approche offre l'avantage de conserver l'état des instances en cours d'exécution (c'est-à-dire sans redémarrage de l'applet ou de la machine virtuelle), contrairement au mécanisme traditionnel de *post-issuance*. Nous montrons que notre approche, basée sur la génération *offline* d'un fichier de différences entre deux versions de l'applet, respecte les contraintes (mémoire, CPU, sécurité) de la carte à puce.

— **Publications** : [CNC4, CNC5, CNC2, CI8, CN4, WI3, WI2, PO1, PO2]

Ahmadou Séré

— **Titre** : "Tissage automatique de contres-mesures pour carte à puce"
— Débutée en Septembre 2007 – soutenue le 23 Septembre 2010
— **Financement** : Bourse du gouvernement Burkinabé
— **Encadrement** : 50% (co-encadré avec le professeur Jean-Louis Lanet)
— **Devenir** : Ingénieur R&D à Morpho (groupe SAGEM) depuis Mars 2011
— **Sujet** : Afin de protéger le *bytecode* contenue dans une carte à puce contre les attaques en faute, nous proposons plusieurs solutions de vérification du code exécuté dans une machine virtuelle Java Card. Ces méthodes se basent sur plusieurs approches comme la détection d'exécution d'opérandes ou le calcul *offline* de propriétés du code (chemins valides ou sommes de contrôle de blocs basiques par exemple) afin de détecter tout changement illégal du flot d'exécution.
— **Publications** : [JIC1, ST4, CI6, CN5, WI4]

9.4 Ingénieurs associés

Grégory Guche

— **Titre** : "Développement de banc-tests pour l'évalition WCET de plate-formes complexe"
— Janvier 2014 – Juin 2014 puis Octobre 2014 – Janvier 2015
— **Financement** : ANR SOBAS puis DGA
— **Direction** : 100%

Hanan Tadmori

— **Titre** : "Outillage du simulateur de l'API INOSSEM"
— Janvier 2013 – Juillet 2013
— **Financement** : FUI INOSSEM
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Pierrick Buret

— **Titre** : "Attaques sur l'ordonnanceur de l'hyperviseur XtratuM"
— Septembre 2012 – Décembre 2012
— **Financement** : ANR SOBAS
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Bhagyalekshmy Narayanan Thampi

— **Titre** : "Machine virtuelle à sécurité adaptative"
— Mai 2012 – Avril 2013
— **Financement** : FUI INOSSEM
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Clément Mazin et Jean-Baptiste Machemie

— **Titre** : “Développement d’un outil d’analyse statique d’applications NFC pour carte à puce”
— Octobre 2011 — Avril 2012
— **Financement** : dispositif “ Détection Innovation Laboratoire ” (DIL) de la région Limousine
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— Les travaux ont débouchés sur la création de l’entreprise Arya Security
— **Publication** : [CI1]

Matthieu Barreaud

— **Titre** : “Fuzzing de protocoles pour cartes à puce”
— Septembre 2010 — Septembre 2011
— **Financement** : projet FUI MECANOS
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publication** : [CNC1, CN2]

Clément Mazin

— **Titre** : “Évaluation de la sécurité de l’hyperviseur OKL4”
— Septembre 2010 — Juin 2011
— **Financement** : projet FUI Tisphanie
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publication** : [WI1]

9.5 Stages

Pierre Graux

— **Titre** : “Backend MBED pour DISQUS”
— Stage de Licence 2, Villeneuve d’Ascq (Juillet – Août 2014)
— **Financement** : Soutien nouvel arrivant LIFL
— **Direction** : 33% (co-direction avec Michael Hauspie et le doctorant Damien Riquet)

Quentin Bergougnoux

— **Titre** : “Étude de SeL4”
— Stage de Master 1, Villeneuve d’Ascq (Juillet – Août 2014)
— **Financement** : Soutien nouvel arrivant LIFL
— **Direction** : 33% (co-direction avec le professeur Gilles Grimaud et le chargé de recherche CNRS David Nowak)

Christophe Baccara

— **Titre** : “Développement de prototypes de PIN rétinien”

— Stage de Master 1, Villeneuve d'Ascq (Juillet – Août 2014)
— **Financement** : Soutien nouvel arrivant LIFL
— **Direction** : 100%

Narjes Jooma

— **Titre** : "Modélisation et preuve sur le mécanisme d'isolation mémoire"
— Stage de fin d'étude, Master 2 TIIR, Villeneuve d'Ascq (Février – Août 2014)
— **Financement** : Soutien nouvel arrivant LIFL
— **Direction** : 33% (co-direction avec le professeur Gilles Grimaud et le chargé de recherche CNRS David Nowak)

Pierrick Buret

— **Titre** : "Sécurité et politique d'ordonnancement de l'hyperviseur XtratuM"
— Stage de fin d'étude, ESIEE, Paris (Février – Août 2012)
— **Financement** : Projet ANR SOBAS
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Jean Dubreuil

— **Titre** : "Application web pour la manipulation de fichier CAP"
— Stage M1 CRYPTIS, Limoges (Juin 2012 – Juillet 2012)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publications** : [CIC2]

Josselin Dolhen

— **Titre** : "Implémentation d'un plug-in Rodin pour générer des tests de vulnérabilité"
— Stage M1 CRYPTIS, Limoges (Juin 2012 – Juillet 2012)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Patrick Silvera

— **Titre** : "Protocole pour la création de réseaux domestiques"
— Stage M2 CRYPTIS, Limoges (Février – Août 2012)
— **Financement** : partenaire ORANGE
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publications** : [CI2]

David Pequegnot

— **Titre** : "Étude de l'hypervisueur Xtratum"
— Stage M2 CRYPTIS, Limoges (Février – Août 2011)
— **Financement** : projet ANR SOBAS
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Jean-Baptiste Machémie

— **Titre** : “Sécurisation des applications Java Card contre les attaques en faute”
— Stage M2 CRYPTIS, Limoges (Avril – Septembre 2011)
— **Financement** : partenaire SFR
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publications** : [CIC4]

Anthony Gautrault

— **Titre** : “Sécurité des communications sous Android”
— Stage M2 CRYPTIS, Limoges (Avril – Septembre 2011)
— **Financement** : partenaire SFR
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Jordan Bouyat

— **Titre** : “Recherche de vulnérabilités sur un hyperviseur par fuzzing et solveur de contraintes”
— Stage IUT 2ème année, Limoges (Avril – Juin 2011)
— **Financement** : projet FUI TISPHANIE
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Tiana Razafindralambo et Yorick Lesecque

— **Titre** : “Analyse statique de bytecode Java Card avec FindBugs”
— Stage M1 CRYPTIS, Limoges (Juin – Juillet 2012)
— **Financement** : projet FUI MECANOS
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Clément Mazin

— **Titre** : “Évaluation de la sécurité de l’hyperviseur OKL4”
— Stage M2 CRYPTIS, Limoges (Février – Août 2010)
— **Financement** : projet FUI INOSSEM
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publications** : [CI4]

Matthieu Barreaud

— **Titre** : “Sécurité de Javacard 3.0”
— Stage M2 CRYPTIS, Limoges (Février – Août 2010)
— **Financement** : projet FUI MECANOS
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)
— **Publications** : [CNC3]

Josselin Dolhen

— **Titre** : “Génération de DIFF pour carte à puce”

— Stage L2 CRYPTIS, Limoges (Juin 2010 -– Juillet 2010)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Louis Bida

— **Titre** : “Portage du protocole de gestion d’APDU sur AT91”
— Stage L2 CRYPTIS, Limoges (Juin 2010 -– Juillet 2010)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Jean-Baptiste Machémie et Lonny Brissac

— **Titre** : “Interface graphique pour un outil d’évaluation d’attaques en faute”
— Stage M1 CRYPTIS, Limoges (Juin 2010 -– Juillet 2010)
— **Financement** : partenaire SFR
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Amaury Gauthier

— **Titre** : “Sécurité des solutions de virtualisation pour téléphones portables”
— Stage M2 CRYPTIS, Limoges (Février – Août 2009)
— **Financement** : projet FUI INOSSEM
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Guillaume Bouffard et Julien Boutet

— **Titre** : “Implémentation de la bibliothèque CapMap”
— Stage M1 CRYPTIS, Limoges (Juin – Juillet 2009)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Ayman Fahs

— **Titre** : “Analyse statique et dynamique de sécurité dans les applications web”
— Stage M2 CRYPTIS, Limoges (Février – Août 2008)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Boulares Ouchene

— **Titre** : “Recherche et analyse des dépendances des variables dans une JavaCard”
— Stage M2 CRYPTIS, Limoges (Février – Août 2008)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Emilie Faugeron et Anthony Dessiatnikoff

— **Titre** : “Évaluation des attaques EMAN dans une Java Card”
— Stage M1 CRYPTIS, Limoges (Juin 2008 – Juillet 2008)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Damien Arcuset et Eric Linke

— **Titre** : “Implémentation de la norme Global Platform Card”
— Stage M1 CRYPTIS, Limoges (Juin 2008 – Juillet 2008)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Ahmadou Séré

— **Titre** : “Évaluation de le sécurité du protocole de routage SIRUP”
— Stage M2 CRYPTIS, Limoges (Février – Août 2007)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

9.6 Projets étudiants

Quentin Bergougnoux

— **Titre** : “Modèle de gestionnaire mémoire”
— Projet M2 TIIR, Villeneuve d’Ascq (Septembre 2014 – Mars 2015)
— **Direction** : 50% (co-direction avec le professeur Gilles Grimaud)

Jérome Vaessen

— **Titre** : “Isolation des accès mémoires sur LEON3”
— PFE, Polytech’Lille, Villeneuve d’Ascq (Septembre 20114 – Mars 2015)
— **Direction** : 50% (co-direction avec le doctorant Pierrick Buret)

Narjes Jooma

— **Titre** : “État de l’art sur les noyaux prouvés”
— Projet M2 TIIR, Villeneuve d’Ascq (Septembre 2013 – Mars 2014)
— **Direction** : 50% (co-direction avec le chargé de recherche David Nowak)

Romain Severin, Mamadou L. Balde, Lylia Tikobaini et Amine Belhociné

— **Titre** : “Génération de grammaire pour fuzzer HTTP”
— Projet M1 CRYPTIS, Limoges (Janvier – Juin 2011)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Nicolas Tarriol, Jérémy Clément, Silvère Cainaud et Amine Belhociné

— **Titre** : “Fuzzing du protocole HTTP sur carte à puce”
— Projet M1 CRYPTIS, Limoges (Janvier – Juin 2011)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Aymerick Savary

— **Titre** : "Portage de SMEWS pour carte Embest STDV710"
— Projet M1 CRYPTIS, Limoges (Janvier – Juin 2010)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Keita Ansoumane

— **Titre** : "Serveur pour la gestion de DIFF d'applications Java"
— Projet M1 CRYPTIS, Limoges (Janvier – Juin 2010)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

David Pequegnot et Julie Rispal

— **Titre** : "Interface Graphique pour la librairie OPAL"
— Projet L3 CRYPTIS, Limoges (Janvier – Juin 2009)
— **Direction** : 50% (co-direction avec le professeur Jean-Louis Lanet)

Chapitre 10

Bibliographie personnelle

Thèses encadrées

[ST1] Pierrick Buret. « Sécurité des hyperviseurs pour satellites ». En cours. Thèse de doct. Université de Limoges, 2015.

[ST2] Nassima Kamel. « Sécurité des cartes à puce à serveur Web embarqué ». Thèse de doct. Université de Limoges, 20 déc. 2012.

[ST3] Agnès Christelle Noubissi. « Mise à jour dynamique pour cartes à puce Java ». Thèse de doct. Université de Limoges, 12 déc. 2011.

[ST4] Ahmadou Séré. « Tissage de contremesures pour machines virtuelles embarquées ». Thèse de doct. Université de Limoges, 23 sept. 2010.

Éditeur

[ED1] Dieter Gollmann, Jean-Louis Lanet et Julien Iguchi-Cartigny, éds. *Smart card research and advanced application : 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010 : proceedings*. Lecture notes in computer science 6035. Berlin ; New York : Springer, 2010. 237 p. isbn : 9783642125096.

Revues internationales avec comité de lecture

[JIC1] Ahmadou Séré, Jean-Louis Lanet et Julien Iguchi-Cartigny. « Evaluation of Countermeasures Against Fault Attacks on Smart Cards ». In : *International journal of Security and Its Applications* 5.2 (2011), p. 49–60.

[JIC2] Julien Iguchi-Cartigny et Jean-Louis Lanet. « Developing a Trojan applets in a smart card ». In : *Journal in Computer Virology* 6.4 (nov. 2010), p. 343–351. issn : 1772-9890, 1772-9904. doi : 10.1007/s11416-009-0135-3. url : http://link.springer.com/10.1007/s11416-009-0135-3.

[JIC3] Julien IGUCHI-CARTIGNY, Pedro M. RUIZ, David SIMPLOT-RYL, Ivan STOJMENOVIĆ et Carmen M. YAGO. « Localized Minimum-Energy Broadcasting for Wireless Multihop Networks with Directional Antennas ». In : *IEEE Transactions on Computers* 58.1 (jan. 2009), p. 120–131. ISSN : 0018-9340. DOI : 10.1109/TC.2008.125. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4585365.

[JIC4] Rami YARED, Xavier DÉFAGO, Julien IGUCHI-CARTIGNY et Matthias WIESMANN. « Collision Prevention Platform for a Dynamic Group of Asynchronous Cooperative Mobile Robots ». In : *Journal of Networks* 2.4 (1er août 2007). ISSN : 1796-2056. DOI : 10.4304/jnw.2.4.28-39. URL : http://www.academypublisher.com/ojs/index.php/jnw/article/view/935.

[JIC5] Julien CARTIGNY, Francois INGELREST, David SIMPLOT-RYL et Ivan STOJMENOVIĆ. « Localized LMST and RNG based minimum-energy broadcast protocols in ad hoc networks ». In : *Ad Hoc Networks* 3.1 (jan. 2005), p. 1–16. ISSN : 15708705. DOI : 10.1016/j.adhoc.2003.09.005. URL : http://linkinghub.elsevier.com/retrieve/pii/S1570870503000684.

[JIC6] Julien CARTIGNY, François INGELREST et David SIMPLOT. « RNG Relay Subset Flooding Protocols in Mobile Ad-Hoc Networks ». In : *International Journal of Foundations of Computer Science* 14.2 (avr. 2003), p. 253–265. ISSN : 0129-0541, 1793-6373. DOI : 10.1142/S0129054103001716. URL : http://www.worldscientific.com/doi/abs/10.1142/S0129054103001716.

[JIC7] Julien CARTIGNY et David SIMPLOT. « Border Node Retransmission Based Probabilistic Broadcast Protocols in Ad-Hoc Networks ». In : *Telecommunication Systems* 22.1 (2003), p. 189–204. ISSN : 1018-4864. DOI : 10.1023/A:1023495021643. URL : http://link.springer.com/article/10.1023/A:1023495021643.

Chapitres d´ouvrage

[CH1] Guillaume BARBU, Guillaume BOUFFARD et Julien IGUCHI-CARTIGNY. « Les aspects sécurité logique des cartes ». In : *la carte à puce, vecteur de système de confiance*. Hermes, 2013.

Conférences internationales avec actes et comités de sélection

[CIC1] Pierrick BURET, Julien IGUCHI-CARTIGNY et Gilles GRIMAUD. « Genetic algorithm for DWCET Evaluation on complex platform ». In : *Proceeding of IEEE International Symposium on Industrial Embedded Systems*. Work-in-progress session. Pise, Italia, juin 2014.

[CIC2] Jean DUBREUIL, Jean-Louis LANET, Guillaume BOUFFARD et Julien CARTIGNY. « Type classification against Fault Enabled Mutant in Java based Smart Card ». In : *AERES - SecSE 2012*. Prague, Czech Republic, 2012, p. 551–556. DOI : 10.1109/AERES.2012.24.

[CIC3] Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Combined Software and Hardware Attacks on the Java Card Control Flow ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Emmanuel Prouff. T. 7079. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, p. 283–296. ISBN : 978-3-642-27256-1. DOI : 10.1007/978-3-642-27257-8_18.

[CIC4] Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet et Julien Cartigny. « SmartCM : A Smart Card Fault Injection Simulator ». In : *Proc. IEEE International Workshop on Information Forensics and Security (WIFS 2011)*. Foz do Iguaçu, Brazil : IEEE, nov. 2011, p. 1–6. ISBN : 978-1-4577-1019-3, 978-1-4577-1017-9, 978-1-4577-1018-6. DOI : 10.1109/WIFS.2011.6123124. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6123124.

[CIC5] David Pequegnot, Laurent Cart-Lamy, Aurelien Thomas, Thibault Tigeon, Julien Iguchi-Cartigny et Jean-Louis Lanet. « A security mechanism to increase confidence in m-transactions ». In : *Proc. 6th IEEE International Conference on Risks and Security of Internet and Systems*. Timisoara, Romania : IEEE, sept. 2011, p. 1–8. ISBN : 978-1-4577-1890-8. DOI : 10.1109/CRiSIS.2011.6061836. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6061836.

[CIC6] Ahmadou Séré, Jean-Louis Lanet et Julien Iguchi-Cartigny. « Checking the Paths to Identify Mutant Application on Embedded Systems ». In : *SecTech 2010, International Conference on Security Technology*. Jeju Island, Korea, 2010, p. 459–468. DOI : 10.1007/978-3-642-17569-5_45.

[CIC7] Carlos Aguilar Melchor, Yves Deswarte et Julien Iguchi-Cartigny. « Closed-Circuit Unobservable Voice over IP ». In : *Proc. 23rd Annual Computer Security Applications Conference (ACSAC 2007)*. Miami, USA : IEEE, déc. 2007, p. 119–128. ISBN : 0-7695-3060-5, 978-0-7695-3060-4. DOI : 10.1109/ACSAC.2007.34. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4412982.

[CIC8] Rami Yared, Julien Cartigny, Xavier Défago et Matthias Wiesmann. « Locality-preserving distributed path reservation protocol for asynchronous cooperative mobile robots ». In : *Proc. Intl. Symp. on Autonomous Decentralized Systems (ISADS'2007)*. Sedona, AZ, USA, 2007, p. 188–195. DOI : 10.1109/ISADS.2007.44.

[CIC9] Julien Cartigny, David Simplot-Ryl et Ivan Stojmenovic. « An Adaptive Localized Scheme for Energy-efficient Broadcasting in Ad hoc Networks with Directional Antennas ». In : *Proc. 9th IFIP Int. Conf. on Personal Wireless Communications (PWC 2004)*. Sous la dir. d'Ignas Niemegeers et Sonia Heemstra de Groot. T. 3260. Lecture Notes in Computer Science. Best paper award. Delft, The Netherlands : Springer-Verlag, Berlin, 2004, p. 399–413. DOI : 10.1007/978-3-540-30199-8_33.

[CIC10] Julien Cartigny et David Simplot. « Border Node Retransmission Based Probabilistic Broadcast Protocols in Ad-Hoc Networks ». In : *Proc. 36th Annual Hawaii Int. Conf. on System Sciences (HICSS-36)*. Hawaii, USA, 2003. DOI : 10.1109/HICSS.2003.1174853.

[CIC11] Julien CARTIGNY, David SIMPLOT et Ivan STOJMENOVIC. « Localized minimum-energy broadcasting in ad-hoc networks ». In : *Proc. IEEE INFOCOM'2003*. T. 3. San Francisco, USA : IEEE, 2003, p. 2210–2217. ISBN : 0-7803-7752-4. DOI : 10.1109/INFCOM.2003.1209241. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1209241.

[CIC12] Julien CARTIGNY, David SIMPLOT et Ivan STOJMENOVIC. « Localized Energy Efficient Broadcast for Wireless Networks with Directional Antennas ». In : *Proc. Mediterranean Ad Hoc Networking Workshop (MED-HOC-NET 2002)*. Sardegna, Italy, 2002.

Workshops internationaux

[WI1] Amaury GAUTHIER, Clement MAZIN, Julien IGUCHI-CARTIGNY et Jean-Louis LANET. « Enhancing Fuzzing Technique for OKL4 Syscalls Testing ». In : *Proc. of the Sixth International Conference on Availability, Reliability and Security (ARES 2011)*. IEEE, août 2011, p. 728–733. ISBN : 978-1-4577-0979-1. DOI : 10.1109/ARES.2011.116. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6046028.

[WI2] Agnes C. NOUBISSI, Julien IGUCHI-CARTIGNY et Jean-Louis LANET. « Hot updates for Java based smart cards ». In : *Proc. of the Third Workshop on Hot Topics in Software Upgrades (HotSwapUp'11)*. IEEE, avr. 2011, p. 168–173. ISBN : 978-1-4244-9195-7. DOI : 10.1109/ICDEW.2011.5767630. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5767630.

[WI3] Agnes C. NOUBISSI, Julien IGUCHI-CARTIGNY et Jean-Louis LANET. « Incremental Dynamic Update for Java-Based Smart Cards ». In : *Proc of The Fifth International Conference on Systems (ICONS 2010)*. IEEE, avr. 2010, p. 110–113. ISBN : 978-1-4244-6231-5. DOI : 10.1109/ICONS.2010.27. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5464138.

[WI4] Ahmadou Al Khary SERE, Julien IGUCHI-CARTIGNY et Jean-Louis LANET. « Automatic detection of fault attack and countermeasures ». In : *Proceedings of the 4th Workshop on Embedded Systems Security*. Grenoble, France, 2009, p. 1–7. ISBN : 9781605587004. DOI : 10.1145/1631716.1631723. URL : http://portal.acm.org/citation.cfm?doid=1631716.1631723%20http://dl.acm.org/citation.cfm?id=1631723.

Conférences internationales

[CI1] Jean-Baptiste MACHEMIE, Clément MAZIN, Jean-Louis LANET et Julien CARTIGNY. « Smart Analyzer an Automatic Rules Compliance Checker ». In : *Chip to Cloud Security*. 2012.

[CI2] Patrick SILVERA, Jean-Louis LANET et Julien IGUCHI-CARTIGNY. « Secure Elements in a Home Networking ». In : *Chip to Cloud Security*. 2012.

[CI3] Anis Bkakria, Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet. « OPAL : an open-source global platform Java Library which includes the remote application management over HTTP ». In : *E-smart 2011*. 2011.

[CI4] Amaury Gauthier, Jean-Louis Lanet, Clément Mazin et Julien Iguchi-Cartigny. « Mobile phone hypervisor testing for vulnerabilities discovery ». In : *Smart Mobility*. 2011.

[CI5] Nassima Kamel, Jean-Louis Lanet et Julien Cartigny. « Static Analysis of Javacard web applications and prevention against Javascript injection code ». In : *The First International Conference on Secure networking and Applications (ICSNA 2011)*. 2011.

[CI6] Ahmadou Séré, Julien Iguchi-Cartigny et Jean-Louis Lanet. « A secure virtual machine for Java Card Platform ». In : *E-smart 2011*. 2011.

[CI7] Nassima Kamel, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Development Methodologies of Java Card Web Applications ». In : *E-smart 2010*. 2010.

[CI8] Agnès Noubissi, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Convergence OSGI-JAVACARD : Fine-grained dynamic update ». In : *E-smart 2010*. 2010.

[CI9] Aziz Benlarbi-Delaï, David Simplot, Julien Cartigny et Jean-Chirstophe Cousin. « Using 3D indoor microwave phase sensitive stereoscopic location system to reduce energy consumption in wireless ad-hoc networks ». In : *Proceedings of the 2nd Smart Objects Conference (sOc'2003)*. 2003.

Conférences nationales avec actes et comités de sélection

[CNC1] Matthieu Barreaud, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Analysis Vulnerabilities in Smart Card Web Server ». In : IEEE, mai 2011, p. 1–5. isbn : 978-1-4577-0735-3. doi : 10.1109/SAR-SSI.2011.5931388. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5931388.

[CNC2] Agnès Noubissi, Julien Cartigny et Jean-Louis Lanet. « EmbedDSU : Un framework de HotSwUp pour cartes à puce Java ». In : *8ème conférence Française en Systèmes d'Exploitation (CFSE)*. 2011.

[CNC3] Nassima Kamel, Jean-Louis Lanet, Julien Cartigny et Matthieu Barreaud. « Perspectives d'utilisation du serveur web embarqué dans la carte à puce Java Card 3 ». In : *Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication 2010 (Majecstic 2010)*. 2010.

[CNC4] Agnès Noubissi, Jean-Louis Lanet et Julien Cartigny. « Carte à puce, vers une durée de vie infinie ». In : *Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication 2009 (Majecstic 2009)*. 2009.

[CNC5] Agnès Noubissi, Ahmadou Séré, Jean-Louis Lanet, Julien Cartigny, Guillaume Bouffard et Julien Boutet. « Carte à puce : attaques et contre-mesures ». In : *Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication 2009 (Majecstic 2009)*. 2009.

Conférences nationales

[CN1] David Pequegnot, Laurent Cart-Lamy, Aurelien Thomas, Thibault Tigeon, Julien Cartigny et Jean-Louis Lanet. « A Security Mechanism to Increase Confidence in M-Transactions ». In : *Journée Thème Emergent NFC*. 6 fév. 2012.

[CN2] Matthieu Barreaud, Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Fuzzing du protocole HTTP sur carte à puce ». In : *Rencontre sur les recherches actuelles en cryptographie 2011 (Crypto'Puces 2011)*. 2011.

[CN3] Amaury Gauthier, Julien Cartigny et Jean-Louis Lanet. « Testing microkernel syscalls for vulnerabilities discovery ». In : *3ème Journée Sécurité des Sysytème et Sûreté des Logiciels*. 2011.

[CN4] Agnès Noubissi, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Secure Hotswapping, un réel problème pour cartes à puce ». In : *Rencontre sur les recherches actuelles en cryptographie 2009 (Crypto'Puces 2009)*. 2009.

[CN5] Ahmadou Séré, Jean-Louis Lanet et Julien Cartigny. « Évaluation de mécanismes de détection d'attaques en fautes sur le tas statique d'une Java Card ». In : *Rencontre sur les recherches actuelles en cryptographie 2009 (Crypto'Puces 2009)*. 2009.

Workshops nationaux

[WN1] Tiana Razafindranlambo, Guillaume Bouffard, Jean-Louis Lanet et Julien Iguchi-Cartigny. « Smart Card Attacks : Enter the matrix ». In : *Journée du GdR SoC-SiP sur le thème "OS et sécurité pour les systèmes embarqués"*. 2012.

Conférences invitées nationales

[CG1] Jean-Louis Lanet et Julien Iguchi-Cartigny. « Évaluation de l'injection de code malicieux dans une Java Card ». In : *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2009)*. 2009.

[CG2] Jean-Louis Lanet et Julien Iguchi-Cartigny. « EMAN : un cheval de troie dans une carte à puce ». In : *Computer and Electronic Security Applications Rendez-vous (César 08)*. 2008.

Posters

[PO1] Agnès Noubissi, Julien Cartigny et Jean-Louis Lanet. « Dynamic upgrade of java card applications ». In : *7ème Conférence Française sur les Systèmes d'Exploitation (CFSE'7)*. 2009.

[PO2] Agnès Noubissi, Jean-Louis Lanet et Julien Cartigny. « Patching running Java Card applications ». In : *Fifth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2009)*. 2009.

Rapports de recherche

[RR1] Julien Cartigny et Xavier Défago. *A Sowing Routing Protocol for Dense Mobile Ad Hoc Networks*. IS-RR-2005-009. School of Information Science, JAIST, 2005.

[RR2] Julien Cartigny, David Simplot et Jean Carle. *Stochastic Flooding Broadcast Protocols in Mobile Wireless Networks*. 2002-3. LIFL, Univ. Lille 1, France, 2002.

[RR3] Julien Cartigny, David Simplot, Jean Carle et Vincent Cordonnier. *Protocole de Diffusion Stochastique dans un Réseau Ad-hoc*. 2002-1. LIFL, Univ. Lille 1, France, 2002.

Thèses

[TH1] Julien Cartigny. « Contributions à la Diffusion dans les Réseaux Ad Hoc ». Thèse de doct. Univ. Lille 1, France, 2003.

Bibliographie

[1] Guillaume Bouffard et Jean-Louis Lanet. « Escalade de privilège dans une carte à puce Java Card ». In : *Symposium sur la sécurité des technologies de l'information et des communications*. SSTIC'2014. Rennes, juin 2014. URL : https://www.sstic.org/2014/presentation/escalade_de_privilege_dans_une_carte_a_puce_java_card/.

[2] Guillaume Bouffard et Jean-Louis Lanet. « Reversing the operating system of a Java based smart card ». In : *Journal of Computer Virology and Hacking Techniques* (juil. 2014). ISSN : 2274-2042. DOI : 10.1007/s11416-014-0218-7. URL : http://link.springer.com/10.1007/s11416-014-0218-7.

[3] Guillaume Bouffard, Jean-Louis Lanet, Michael Lackner et Johannes Loinig. « Heap Hop ! The Heap Is Also Vulnerable ». In : *CARDIS'2014*. To be published. Nov. 2014.

[4] Guillaume Bouffard, Jean-Louis Lanet, Rokia Lamrani, Ranim Chakra, Afef Mestiri et Mohammed Monsif. « Memory Forensics of a Java Card Dump ». In : *CARDIS'2014*. To be published. Nov. 2014.

[5] Andrei Costin, Jonas Zaddach, Aurélien Francillon et Davide Balzarotti. « A Large Scale Analysis of the Security of Embedded Firmwares ». In : *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. Août 2014.

[6] Emilie Faugeron. « Manipulating the Frame Information with an Underflow Attack ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Aurélien Francillon et Pankaj Rohatgi. T. 8419. Cham : Springer International Publishing, 2014, p. 140–151. ISBN : 978-3-319-08301-8, 978-3-319-08302-5. URL : http://link.springer.com/10.1007/978-3-319-08302-5_10 (visité le 06/10/2014).

[7] Jevgeni Kabanov et Varmo Vene. « A thousand years of productivity : the JRebel story ». In : *Software : Practice and Experience* 44.1 (jan. 2014), p. 105–127. ISSN : 00380644. DOI : 10.1002/spe.2158. URL : http://doi.wiley.com/10.1002/spe.2158 (visité le 07/10/2014).

[8] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski et Gernot Heiser. « Comprehensive formal verification of an OS microkernel ». In : *ACM Transactions on Computer Systems* 32.1 (1er fév. 2014), p. 1–70. ISSN : 07342071. DOI : 10.1145/2560537. URL : http://dl.acm.org/citation.cfm?doid=2584468.2560537 (visité le 10/09/2014).

[9] Pierre Le Pallec, Sophie Diallo, Thierry Simon, Ahmad Saif, Olivier Briot, Patrick Picard, laurent Fourreau, Sylvie Camus, Michael Bensimon, Jérome Devisme, Maryline Eznack et Saïd Benlaadam. *Cardlet Development Guidelines (v2.4)*. Sous la dir. de Grégory Fraisse et Laurent Kisielewski. 24 juin 2014.

[10] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco et David Brumley. « Optimizing Seed Selection for Fuzzing ». In : *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA : USENIX Association, août 2014, p. 861–875. isbn : 978-1-931971-15-7. url : https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert.

[11] Michal Zalewski. *Binary fuzzing strategies : what works, what doesn't*. lcamtuf's blog. 8 août 2014. url : http://lcamtuf.blogspot.fr/2014/08/binary-fuzzing-strategies-what-works.html.

[12] Guillaume Bouffard, Tom Khefif, Jean-Louis Lanet, Ismael Kane et Sergio Casanova Salvia. « Accessing secure information using export file fraudulence ». In : IEEE, oct. 2013, p. 1–5. isbn : 978-1-4799-3488-1. doi : 10.1109/CRiSIS.2013.6766346. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6766346 (visité le 06/10/2014).

[13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill et Robert Bowdidge. « Why don't software developers use static analysis tools to find bugs ? » In : IEEE, mai 2013, p. 672–681. isbn : 978-1-4673-3076-3, 978-1-4673-3073-2. doi : 10.1109/ICSE.2013.6606613. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6606613 (visité le 11/09/2014).

[14] Karsten Nohl. « Rooting SIM cards ». In : Blackhat USA 2013. Las Vegas, USA, juil. 2013. url : http://www.blackhat.com/us-13/briefings.html#Nohl.

[15] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter et Gunter Saake. « Flexible runtime updates of Java applications ». In : *Software : Practice and Experience* 43.2 (fév. 2013), p. 153–185. issn : 00380644. doi : 10.1002/spe.2107. url : http://doi.wiley.com/10.1002/spe.2107 (visité le 07/10/2014).

[16] Bruno Robisson et Hughes Thiébeauld. « Attaques physiques ». In : *Les cartes à puce*. Sous la dir. de Pierre Paradinas et Samia Bouzefrane. Informatique et Systèmes d'Information. Lavoisier Hermes, sept. 2013. isbn : 9782746239135.

[17] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek et Armando Solar-Lezama. « Towards optimization-safe systems : analyzing the impact of undefined behavior ». In : ACM Press, 2013, p. 260–275. isbn : 9781450323888. doi : 10.1145/2517349.2522728. url : http://dl.acm.org/citation.cfm?doid=2517349.2522728 (visité le 19/10/2014).

[18] François-xavier Aranda et Jean-Louis Lanet. « Smart Card Reverse-Engineering Binary Code Execution Using Side-Channel Analysis ». In : (2012).

[19] Samiya HAMADOUCHE, Guillaume BOUFFARD, Jean-Louis LANET, Bruno DORSEMAINE, Bastien NOUHANT, Alexandre MAGLOIRE et Arnaud REYGNAUD. « Subverting Byte Code Linker service to characterize Java Card API ». In : *Proceedings SARSSI 2012*. Cabourg, France, 2012, p. 75–81. URL : http://hal.archives-ouvertes.fr/hal-00937333.

[20] Christian HOLLER, Kim HERZIG et Andreas ZELLER. « Fuzzing with Code Fragments ». In : *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA : USENIX, 2012, p. 445–458. ISBN : 978-931971-95-9. URL : https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[21] Tiana RAZAFINDRALAMBO, Guillaume BOUFFARD et Jean-Louis LANET. « A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard ». In : *Data and Applications Security and Privacy XXVI*. Sous la dir. de Nora CUPPENS-BOULAHIA, Frédéric CUPPENS et Joaquin GARCIA-ALFARO. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Doug TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 7371. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 122–128. ISBN : 978-3-642-31539-8, 978-3-642-31540-4. URL : http://link.springer.com/10.1007/978-3-642-31540-4_10 (visité le 06/10/2014).

[22] Matthew VAN GUNDY et Hao CHEN. « Noncespaces : Using randomization to defeat cross-site scripting attacks ». In : *Computers and Security* 31.4 (juin 2012), p. 612–628. ISSN : 01674048. DOI : 10.1016/j.cose.2011.12.004. (Visité le 20/09/2014).

[23] Loïc ZUSSA, Jean-Max DUTERTRE, Jessy CLÉDIÈRE, Bruno ROBISSON et Assia TRIA. « Investigation of timing constraints violation as a fault injection means ». In : *pas encore paru*. Avignon, France, 2012, pas encore paru. URL : http://hal-emse.ccsd.cnrs.fr/emse-00742652.

[24] Guillaume BARBU, Guillaume DUC et Philippe HOOGVORST. « Java Card Operand Stack : Fault Attacks, Combined Attacks and Countermeasures ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Emmanuel PROUFF. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Doug TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 7079. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 297–313. ISBN : 978-3-642-27256-1, 978-3-642-27257-8. URL : http://link.springer.com/10.1007/978-3-642-27257-8_19 (visité le 06/10/2014).

[25] Guillaume BARBU et Hugues THIEBEAULD. « Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0 - ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Emmanuel PROUFF. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Doug TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 7079. Berlin,

Heidelberg : Springer Berlin Heidelberg, 2011, p. 18–33. ISBN : 978-3-642-27256-1, 978-3-642-27257-8. URL : http://link.springer.com/10.1007/978-3-642-27257-8_2 (visité le 06/10/2014).

[26] Loïc DUFLOT, Yves-alexis PEREZ et Benjamin MORIN. « What If You Can't Trust Your Network Card ? » In : *Recent Advances in Intrusion Detection*. Sous la dir. de Robin SOMMER, Davide BALZAROTTI et Gregor MAIER. T. 6961. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, p. 378–397. ISBN : 978-3-642-23643-3. URL : http://dx.doi.org/10.1007/978-3-642-23644-0_20%20http://link.springer.com/chapter/10.1007/978-3-642-23644-0_20.

[27] Mathieu JAUME, Valérie VIET TRIEM TONG et Ludovic MÉ. « Flow Based Interpretation of Access Control : Detection of Illegal Information Flows ». In : *Information Systems Security*. Sous la dir. de Sushil JAJODIA et Chandan MAZUMDAR. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Doug TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 7093. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 72–86. ISBN : 978-3-642-25559-5, 978-3-642-25560-1. URL : http://link.springer.com/10.1007/978-3-642-25560-1_5 (visité le 16/10/2014).

[28] Julien LANCIA. « Un framework de fuzzing pour cartes 'a puce : application aux protocoles EMV ». In : *SSTIC 2011*. Rennes, France, 2011. URL : https://www.sstic.org/2011/presentation/framework_de_fuzzing_pour_cartes_a_puce/.

[29] Ikhwan LEE, Mehmet BASOGLU, Michael SULLIVAN, Doe Hyun YOON, Larry KAPLAN et Mattan EREZ. *Survey of Error and Fault Detection Mechanisms*. technical report TH-LPH-2011-02. Avr. 2011. URL : http://lph.ece.utexas.edu/merez/uploads/MattanErez/detection_mechanisms_TR_LPH_2011_002.pdf.

[30] Fernand LONE SANG, Loïc DUFLOT, Vincent NICOMETTE et Yves DESWARTE. « Attaques DMA peer-to-peer et contremesures ». In : *SSTIC 2011*. Juin 2011.

[31] Joel WEINBERGER, Prateek SAXENA, Devdatta AKHAWE, Matthew FINIFTER, Richard SHIN et Dawn SONG. « A Systematic Analysis of XSS Sanitization in Web Application Frameworks ». In : *Computer Security – ESORICS 2011*. Sous la dir. de Vijay ATLURI et Claudia DIAZ. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Doug TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 6879. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 150–171. ISBN : 978-3-642-23821-5, 978-3-642-23822-2. URL : http://link.springer.com/10.1007/978-3-642-23822-2_9 (visité le 20/09/2014).

[32] J. ANDERSSON, J. GAISLER et R. WEIGAND. « Next Generation Multipurpose Microprocessor ». In : *Data Systems In Aerospace 2010 (DASIA2010)*. 2010.

[33] Piotr Bania. « Security Mitigations for Return-Oriented Programming Attacks ». In : *CoRR* abs/1008.4099 (2010). URL : http://arxiv.org/abs/1008.4099.

[34] Guillaume Barbu, Hugues Thiebeauld et Vincent Guerin. « Attacks on Java Card 3.0 Combining Fault and Logical Attacks ». In : *Smart Card Research and Advanced Application*. Sous la dir. de Dieter Gollmann, Jean-Louis Lanet et Julien Iguchi-Cartigny. Réd. par David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi et Gerhard Weikum. T. 6035. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 148–163. ISBN : 978-3-642-12509-6, 978-3-642-12510-2. URL : http://link.springer.com/10.1007/978-3-642-12510-2_11 (visité le 06/10/2014).

[35] *Bearer independent protocol (bip)*. Fév. 2010.

[36] Simon Duquennoy et Gilles Grimaud. « Efficient Web Requests Scheduling Considering Resources Sharing ». In : IEEE, août 2010, p. 410–412. ISBN : 978-1-4244-8181-1. DOI : 10.1109/MASCOTS.2010.53. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5581611 (visité le 16/10/2014).

[37] Émilie Faugeron. « How helpful can be the security assessment of TOE associated tools during a Common Criteria evaluation ». In : *11th International Common Criteria Conference (ICCC 2010)*. 2010.

[38] Émilie Faugeron et Sébastien Valette. « How to hoax an off-card verifier ». In : *E-smart 2010*. 2010.

[39] Gernot Heiser et Ben Leslie. « The OKL4 microvisor : convergence point of microkernels and hypervisors ». In : ACM Press, 2010, p. 19. ISBN : 9781450301954. DOI : 10.1145/1851276.1851282. URL : http://portal.acm.org/citation.cfm?doid=1851276.1851282 (visité le 19/10/2014).

[40] Éric Lacombe, Fernand Lone Sang, Vincent Nicomette et Yves Desswarte. « Analyse de l'efficacité du service fourni par une IOMMU ». In : *SSTIC 2010*. Juin 2010.

[41] W. Rankl. *Smart card handbook*. 4th ed. Chichester, West Sussex, U.K : Wiley, 2010. 1043 p. ISBN : 9780470743676.

[42] Thomas Würthinger, Christian Wimmer et Lukas Stadler. « Dynamic code evolution for Java ». In : ACM Press, 2010, p. 10. ISBN : 9781450302692. DOI : 10.1145/1852761.1852764. URL : http://portal.acm.org/citation.cfm?doid=1852761.1852764 (visité le 07/10/2014).

[43] ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*. PRD29-GENC-009492C. ARM Limited, 2009. URL : http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[44] Erika CHIN et David WAGNER. « Efficient character-level taint tracking for Java ». In : ACM Press, 2009, p. 3. ISBN : 9781605587899. DOI : 10.1145/1655121.1655125. URL : http://portal.acm.org/citation.cfm?doid=1655121.1655125 (visité le 20/09/2014).

[45] Loïc DUFLOT. « CPU bugs, CPU backdoors and consequences on security ». In : *Journal in computer virology* 5.2 (2009), p. 91–104. ISSN : 1772-9890. DOI : 10.1007/s11416-008-0109-x. URL : http://link.springer.com/article/10.1007/s11416-008-0109-x.

[46] Manuel EGELE, Engin KIRDA et Christopher KRUEGEL. « Mitigating Drive-By Download Attacks : Challenges and Open Problems ». In : *iNetSec 2009 – Open Research Problems in Network Security*. Sous la dir. de Jan CAMENISCH et Dogan KESDOGAN. T. 309. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 52–62. ISBN : 978-3-642-05436-5, 978-3-642-05437-2. URL : http://link.springer.com/10.1007/978-3-642-05437-2_5 (visité le 19/10/2014).

[47] Allan Raundahl GREGERSEN, Douglas SIMON et Bo Nørregaard JØRGENSEN. « Towards a dynamic-update-enabled JVM ». In : ACM Press, 2009, p. 1–7. ISBN : 9781605585482. DOI : 10.1145/1562860.1562862. URL : http://portal.acm.org/citation.cfm?doid=1562860.1562862 (visité le 07/10/2014).

[48] « Method of securely running an application ». EP2043017 A1. Laurent GAUTERON. 1er avr. 2009. URL : https://www.google.com/patents/EP2043017A1.

[49] William ROBERTSON et Giovanni VIGNA. « Static Enforcement of Web Application Integrity Through Strong Typing ». In : *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM'09. Berkeley, CA, USA : USENIX Association, 2009, p. 283–298. URL : http://dl.acm.org/citation.cfm?id=1855768.1855786.

[50] Suriya SUBRAMANIAN, Michael HICKS et Kathryn S. MCKINLEY. « Dynamic software updates : a VM-centric approach ». In : ACM Press, 2009, p. 1. ISBN : 9781605583921. DOI : 10.1145/1542476.1542478. URL : http://portal.acm.org/citation.cfm?doid=1542476.1542478 (visité le 07/10/2014).

[51] Joshua BLOCH. *Effective Java (2Nd Edition) (The Java Series)*. 2e éd. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2008. ISBN : 0321356683, 9780321356680.

[52] Wojciech MOSTOWSKI et Erik POLL. « Malicious code on Java Card smartcards : Attacks and countermeasures ». In : *Smart Card Research and Advanced Applications* (2008). URL : http://link.springer.com/chapter/10.1007/978-3-540-85893-5_1.

[53] Ari TAKANEN. *Fuzzing for software security testing and quality assurance*. Avec la coll. de Jared D. DEMOTT et Charles MILLER. Artech House information security and privacy series. Norwood, MA : Artech House, 2008. 287 p. ISBN : 1596932147.

[54] Gilles BARTHE, Lilian BURDY, Julien CHARLES, Benjamin GRÉGOIRE, Marieke HUISMAN, Jean-Louis LANET, Mariela PAVLOVA et Antoine REQUET. « JACK — A Tool for Validation of Security and Behaviour of Java Applications ». In : *Formal Methods for Components and Objects*. Sous la dir. de Frank S. de BOER, Marcello M. BONSANGUE, Susanne GRAF et Willem-Paul de ROEVER. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Doug TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 4709. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 152–174. ISBN : 978-3-540-74791-8, 978-3-540-74792-5. URL : http://link.springer.com/10.1007/978-3-540-74792-5_7 (visité le 16/10/2014).

[55] Dorina GHINDICI, Gilles GRIMAUD et Isabelle SIMPLOT-RYL. « An Information Flow Verifier for Small Embedded Systems ». In : *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*. Sous la dir. de Damien SAUVERON, Konstantinos MARKANTONAKIS, Angelos BILAS et Jean-Jacques QUISQUATER. T. 4462. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 189–201. ISBN : 978-3-540-72353-0, 978-3-540-72354-7. URL : http://link.springer.com/10.1007/978-3-540-72354-7_16 (visité le 19/09/2014).

[56] Charlie MILLER et Zachary NJ PETERSON. *Analysis of mutation and generation-based fuzzing*. 2007. URL : http://www.securityevaluators.com/files/papers/analysisfuzzing.pdf.

[57] Wojciech MOSTOWSKI et Erik POLL. « Testing the Java Card Applet Firewall ». In : (2007), p. 1–9. URL : https://vsm.cs.utwente.nl/%20mostowskiwi/papers/firewall2007.pdf.

[58] Stephen SOLTESZ, Herbert POTZL, Marc E. FIUCZYNSKI, Andy BAVIER et Larry PETERSON. « Container-based operating system virtualization : a scalable, high-performance alternative to hypervisors ». In : *ACM SIGOPS Operating Systems Review* 41.3 (1er juin 2007), p. 275. ISSN : 01635980. DOI : 10.1145/1272998.1273025. URL : http://portal.acm.org/citation.cfm?doid=1272998.1273025 (visité le 19/10/2014).

[59] Michael SUTTON. *Fuzzing : brute force vulnerabilty discovery*. Avec la coll. d'Adam GREENE et Pedram AMINI. Upper Saddle River, NJ : Addison-Wesley, 2007. 543 p. ISBN : 0321446119.

[60] Dennis VERMOEN, Marc WITTEMAN et Georgi N. GAYDADJIEV. « Reverse Engineering Java Card Applets Using Power Analysis ». In : (2007), p. 138–149. DOI : 10.1007/978-3-540-72354-7_12.

[61] Keith ADAMS et Ole AGESEN. « A comparison of software and hardware techniques for x86 virtualization ». In : *ACM SIGARCH Computer Architecture News* 34.5 (20 oct. 2006), p. 2. ISSN : 01635964. DOI : 10.1145/1168919.1168860. (Visité le 10/09/2014).

[62] Hagai BAR-EL, Hamid CHOUKRI, David NACCACHE, Michael TUNSTALL et Claire WHELAN. « The Sorcerer's Apprentice Guide to Fault Attacks ». In : *Proceedings of the IEEE* 94.2 (2006), p. 370–382. ISSN : 0018-9219. DOI : 10.1109/JPROC.2005.862424.

[63] Jacques FOURNIER et Michael TUNSTALL. « Cache Based Power Analysis Attacks on AES ». In : *Information Security and Privacy*. Sous la dir. de Lynn Margaret BATTEN et Reihaneh SAFAVI-NAINI. Réd. par David HUTCHISON, Takeo KANADE, Josef KITTLER, Jon M. KLEINBERG, Friedemann MATTERN, John C. MITCHELL, Moni NAOR, Oscar NIERSTRASZ, C. PANDU RANGAN, Bernhard STEFFEN, Madhu SUDAN, Demetri TERZOPOULOS, Dough TYGAR, Moshe Y. VARDI et Gerhard WEIKUM. T. 4058. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 17–28. ISBN : 978-3-540-35458-1, 978-3-540-35459-8. URL : http://link.springer.com/10.1007/11780656_2 (visité le 26/09/2014).

[64] Kevin HENRY. « A crash overview of groovy ». In : *Crossroads* 12.3 (1er mai 2006), p. 5–5. ISSN : 15284972. DOI : 10.1145/1144366.1144371. URL : http://portal.acm.org/citation.cfm?doid=1144366.1144371 (visité le 07/10/2014).

[65] Jorrit N. HERDER, Herbert BOS, Ben GRAS, Philip HOMBURG et Andrew S. TANENBAUM. « MINIX 3 : a highly reliable, self-repairing operating system ». In : *ACM SIGOPS Operating Systems Review* 40.3 (1er juil. 2006), p. 80. ISSN : 01635980. DOI : 10.1145/1151374.1151391. URL : http://portal.acm.org/citation.cfm?doid=1151374.1151391 (visité le 19/10/2014).

[66] Engelbert HUBBERS, Wojciech MOSTOWSKI et Erik POLL. « Tearing Java Cards ». In : *Proceedings, e-Smart* (2006), p. 1–21. URL : http://www.cs.ru.nl/E.Poll/publications/esmart06.pdf.

[67] Vinay M. IGURE, Sean A. LAUGHTER et Ronald D. WILLIAMS. « Security issues in SCADA networks ». In : *Computers & Security* 25.7 (oct. 2006), p. 498–506. ISSN : 01674048. DOI : 10.1016/j.cose.2006.03.001. URL : http://linkinghub.elsevier.com/retrieve/pii/S0167404806000514 (visité le 19/10/2014).

[68] Mayur NAIK, Alex AIKEN et John WHALEY. « Effective static race detection for Java ». In : *ACM SIGPLAN Notices* 41.6 (11 juin 2006), p. 308. ISSN : 03621340. DOI : 10.1145/1133255.1134018. URL : http://portal.acm.org/citation.cfm?doid=1133255.1134018 (visité le 16/10/2014).

[69] AMD. *Secure Virtual Machine Architecture Reference Manual*. Advanced Micro Devices, mai 2005. URL : http://www.mimuw.edu.pl/%20vincent/lecture6/sources/amd-pacifica-specification.pdf.

[70] *Application Programming Interface Java Card Platform, Version 2.2.2*. 2005.

[71] Lilian BURDY, Yoonsik CHEON, David R. COK, Michael D. ERNST, Joseph R. KINIRY, Gary T. LEAVENS, K. Rustan M. LEINO et Erik POLL. « An overview of JML tools and applications ». In : *International Journal on Software Tools for Technology Transfer* 7.3 (juin 2005), p. 212–232. ISSN : 1433-2779, 1433-2787. DOI : 10.1007/s10009-004-0167-4. URL : http://link.springer.com/10.1007/s10009-004-0167-4 (visité le 16/10/2014).

[72] Engelbert HUBBERS et Erik POLL. « Transactions and non-atomic API methods in Java Card : specification ambiguity and strange implementation behaviours ». In : *cs.ru.nl* (nov. 2005), p. 1–22. URL : http://cs.ru.nl/E.Poll/papers/acna_new.pdf.

[73] Eduard de Jong, Prof dr Pieter Hartel, Patrice Peyret et Peter Cattaneo. *Java Card : an analysis of the most successful smart card operating system to date*. technical report CTIT-TR-05-50. Version 1.1. Enschede, The Netherlands : University of Twente, Centre for Telematics et Information Technology (CTIT), 2005. URL : http://doc.utwente.nl/54542/.

[74] *Runtime Environment Specification Java Card Platform, Version 2.2.2*. 2005.

[75] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung et L. Smith. « Intel virtualization technology ». In : *Computer* 38.5 (mai 2005), p. 48–56. ISSN : 0018-9162. DOI : 10.1109/MC.2005.163. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1430631 (visité le 11/09/2014).

[76] *Virtual Machine Specification Java Card Platform, Version 2.2.2*. 2005.

[77] Nadia Bel Hadj Aissa, Christophe Rippert, Damien Deville et Gilles Grimaud. « A Distributed WCET Computation Scheme for Smart Card Operating Systems ». In : *4th International Workshop on Worst-Case Execution Time Analysis, in association with the 16th Euromicro ECRTS conference*. Catania, Italie, juin 2004.

[78] David Hovemeyer et William Pugh. « Finding bugs is easy ». In : *ACM SIGPLAN Notices* 39.12 (1er déc. 2004), p. 92. ISSN : 03621340. DOI : 10.1145/1052883.1052895. URL : http://portal.acm.org/citation.cfm?doid=1052883.1052895 (visité le 20/09/2014).

[79] Engelbert Hubbers et Erik Poll. *Reasoning about card tears and transactions in JAVA CARD*. 2004. 1–15. URL : http://link.springer.com/chapter/10.1007/978-3-540-24721-0_8.

[80] Martin Otto. « Fault Attacks and Countermeasures ». Dissertation. Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, 2004.

[81] Mehdi Akkar, Louis Goubin et Olivier Ly. *Automatic Integration of Counter-Measures Against Fault Injection Attacks*. 2003. URL : http://www.labri.fr/perso/ly/publications/cfed.pdf.

[82] Paul Barham, Boris Dragovic et Keir Fraser. « Xen and the art of virtualization ». In : *ACM SIGOPS* 37 (2003), p. 164. ISSN : 01635980. DOI : 10.1145/1165389.945462. URL : http://portal.acm.org/citation.cfm?id=945462%20http://dl.acm.org/citation.cfm?id=945462.

[83] Damien Deville, Antoine Galland, Gilles Grimaud et Sébastien Jean. « Assessing the Future of Smart Card Operating Systems ». In : (2003).

[84] Sudhakar Govindavajhala et Andrew W. Appel. « Using Memory Errors to Attack a Virtual Machine ». In : *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. SP '03. Washington, DC, USA : IEEE Computer Society, 2003, p. 154–. ISBN : 0-7695-1940-7. URL : http://dl.acm.org/citation.cfm?id=829515.830563.

[85] Gilles Grimaud et Jean-jacques Vandewalle. « Introducing Research Issues for Next Generation Java-based Smart Card Platforms ». In : (2003), p. 1–4.

[86] Stefan MANGARD. « A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion ». In : *Information Security and Cryptology — ICISC 2002*. Sous la dir. de Pil Joong LEE et Chae Hoon LIM. Réd. par Gerhard GOOS, Juris HARTMANIS et Jan van LEEUWEN. T. 2587. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 343–358. ISBN : 978-3-540-00716-6, 978-3-540-36552-5. URL : http://link.springer.com/10.1007/3-540-36552-4_24 (visité le 25/09/2014).

[87] Konstantinos MARKANTONAKIS et Keith MAYES. « An overview of the GlobalPlatform smart card specification ». In : *Information Security Technical Report* 8.1 (mar. 2003), p. 17–29. ISSN : 13634127. DOI : 10.1016/S1363-4127(03)00103-1. URL : http://linkinghub.elsevier.com/retrieve/pii/S1363412703001031 (visité le 23/09/2014).

[88] Pat NIEMEYER et Daniel LEUCK. *Beanshell - lightweight scripting for Java*. 2003.

[89] OSGi ALLIANCE. *OSGi service platform : release 3, March 2003*. Amsterdam ; Washington, DC : IOS Press : Ohmsha, 2003. 586 p. ISBN : 1586033115.

[90] Sergei P. SKOROBOGATOV et Ross J. ANDERSON. « Optical Fault Induction Attacks ». In : *Cryptographic Hardware and Embedded Systems - CHES 2002*. Sous la dir. de Burton S. KALISKI et Christof PAAR. Réd. par Gerhard GOOS, Juris HARTMANIS et Jan van LEEUWEN. T. 2523. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 2–12. ISBN : 978-3-540-00409-7, 978-3-540-36400-9. URL : http://link.springer.com/10.1007/3-540-36400-5_2 (visité le 26/09/2014).

[91] Gilles BARTHE, Guillaume DUFAY, Line JAKUBIEC et Simão Melo de SOUSA. « A Formal Correspondence Between Offensive and Defensive JavaCard Virtual Machines ». In : *Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*. VMCAI '02. London, UK, UK : Springer-Verlag, 2002, p. 32–45. ISBN : 3-540-43631-6. URL : http://dl.acm.org/citation.cfm?id=646541.696190.

[92] Gabriel BIZZOTTO et Gilles GRIMAUD. « Practical java card bytecode compression ». In : *Proceedings of RENPAR14/ASF/SYMPA*. 2002.

[93] Mikhail DMITRIEV. « Application of the HotSwap technology to advanced profiling ». In : *Proceedings of the ECOOP*. T. 2. 2002.

[94] R. LAI. « A survey of communication protocol testing ». In : *Journal of Systems and Software* 62.1 (mai 2002), p. 21–46. ISSN : 01641212. DOI : 10.1016/S0164-1212(01)00132-7. URL : http://linkinghub.elsevier.com/retrieve/pii/S0164121201001327 (visité le 03/10/2014).

[95] A. ORSO, A. RAO et M.J. HARROLD. « A technique for dynamic updating of Java software ». In : IEEE Comput. Soc, 2002, p. 649–658. ISBN : 0-7695-1819-2. DOI : 10.1109/ICSM.2002.1167829. URL : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1167829 (visité le 07/10/2014).

[96] Mehdi-Laurent Akkar et Christophe Giraud. « An Implementation of DES and AES, Secure against Some Attacks ». In : *Cryptographic Hardware and Embedded Systems — CHES 2001*. T. 2162. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, p. 309–318. isbn : 978-3-540-42521-2, 978-3-540-44709-2. url : http://link.springer.com/10.1007/3-540-44709-1_26 (visité le 26/09/2014).

[97] Joachim van den Berg et Bart Jacobs. « The loop Compiler for Java and JML ». In : *Tools and Algorithms for the Construction and Analysis of Systems*. Sous la dir. de Tiziana Margaria et Wang Yi. Réd. par Gerhard Goos, Juris Hartmanis et Jan van Leeuwen. T. 2031. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, p. 299–312. isbn : 978-3-540-41865-8, 978-3-540-45319-2. url : http://link.springer.com/10.1007/3-540-45319-9_21 (visité le 16/10/2014).

[98] P. Bieber, J. Cazin, V. Wiels, G. Zanon, El Marouan, P. Girard et J.-L. Lanet. « The PACAP Prototype : A Tool for Detecting Java Card Illegal Flow ». In : *Java on Smart Cards :Programming and Security*. Sous la dir. d'Isabelle Attali et Thomas Jensen. Réd. par Gerhard Goos, Juris Hartmanis et Jan van Leeuwen. T. 2041. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, p. 25–37. isbn : 978-3-540-42167-2, 978-3-540-45165-5. url : http://link.springer.com/10.1007/3-540-45165-X_3 (visité le 16/10/2014).

[99] Jean-Jacques Quisquater et David Samyde. « ElectroMagnetic Analysis (EMA) : Measures and Counter-measures for Smart Cards ». In : *Smart Card Programming and Security*. Sous la dir. d'Isabelle Attali et Thomas Jensen. Réd. par Gerhard Goos, Juris Hartmanis et Jan van Leeuwen. T. 2140. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, p. 200–210. isbn : 978-3-540-42610-3, 978-3-540-45418-2. url : http://link.springer.com/10.1007/3-540-45418-7_17 (visité le 25/09/2014).

[100] Jesper Andersson et Tobias Ritzau. « Dynamic code update in JDrums ». In : *Proceedings of the ICSE'00 Workshop on Software Engineering for Wearable and Pervasive Computing*. Citeseer, 2000.

[101] Zhiqun Chen. *Java Card technology for Smart Cards : architecture and programmer's guide*. The Java series. Boston : Addison-Wesley, 2000. 368 p. isbn : 0201703297.

[102] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater et Jean-Louis Willems. « A Practical Implementation of the Timing Attack ». In : *Smart Card Research and Applications*. Sous la dir. de Jean-Jacques Quisquater et Bruce Schneier. Réd. par Gerhard Goos, Juris Hartmanis et Jan van Leeuwen. T. 1820. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 167–182. isbn : 978-3-540-67923-3, 978-3-540-44534-0. url : http://link.springer.com/10.1007/10721064_15 (visité le 26/09/2014).

[103] Gilles Grimaud. « CAMILLE : un Système d'Exploitation Ouvert pour Carte à Microprocesseur ». Thèse de doct. France : Université de Lille 1, déc. 2000.

[104] Rita Mayer-Sommer. « Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards ». In : *Cryptographic Hardware and Embedded Systems — CHES 2000*. Réd. par Gerhard Goos, Juris Hartmanis et Jan van Leeuwen. T. 1965. Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 78–92. isbn : 978-3-540-41455-1, 978-3-540-44499-2. url : http://link.springer.com/10.1007/3-540-44499-8_6 (visité le 25/09/2014).

[105] Barry Redmond et Vinny Cahill. « Iguana/J : Towards a dynamic and efficient reflective architecture for Java ». In : *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*. 2000.

[106] Jim Rees et Peter Honeyman. « Webcard : A Java Card Web Server ». In : *Smart Card Research and Advanced Applications*. Sous la dir. de Josep Domingo-Ferrer, David Chan et Anthony Watson. Boston, MA : Springer US, 2000, p. 197–207. isbn : 978-1-4757-6526-7, 978-0-387-35528-3. url : http://link.springer.com/10.1007/978-0-387-35528-3_11 (visité le 03/10/2014).

[107] C. Cowan, F. Wagle, Calton Pu, S. Beattie et J. Walpole. « Buffer overflows : attacks and defenses for the vulnerability of the decade ». In : t. 2. IEEE Comput. Soc, 1999, p. 119–129. isbn : 0-7695-0490-6. doi : 10.1109/DISCEX.2000.821514. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=821514 (visité le 06/10/2014).

[108] Paul Kocher, Joshua Jaffe et Benjamin Jun. « Differential Power Analysis ». In : *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology* (1999), p. 388–397.

[109] R. Fielding, J. Gettys, J. Mogul, H. Frystyk et T. Berners-Lee. *RFC 2068 : Hypertext Transfer Protocol — HTTP/1.1*. Jan. 1997. url : ftp://ftp.internic.net/rfc/rfc2068.txt.

[110] D. Gupta, P. Jalote et G. Barua. « A formal framework for on-line software version change ». In : *IEEE Transactions on Software Engineering* 22.2 (fév. 1996), p. 120–131. issn : 00985589. doi : 10.1109/32.485222. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=485222 (visité le 07/10/2014).

[111] Paul C. Kocher. « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ». In : *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '96. London, UK, UK : Springer-Verlag, 1996, p. 104–113. isbn : 3-540-61512-1. url : http://dl.acm.org/citation.cfm?id=646761.706156.

[112] J. F. Ziegler et W. A. Lanford. « Effect of Cosmic Rays on Computer Memories ». In : *Science* 206.4420 (16 nov. 1979), p. 776–788. issn : 0036-8075, 1095-9203. doi : 10.1126/science.206.4420.776. url : http://www.sciencemag.org/cgi/doi/10.1126/science.206.4420.776 (visité le 07/10/2014).

[113] Peter Van Emde Boas. *The correspondence between Donald E. Knuth and Peter van Emde Boas on priority deques during the spring of 1977*. 1977. url : https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf.

[114] Henry Gordon Rice. « Classes of recursively enumerable sets and their decision problems ». In : *Transactions of the American Mathematical Society* (1953), p. 358–366.

Première partie

Annexes

Annexe A

CV détaillé

Julien Iguchi-Cartigny

État civil

Julien Iguchi-Cartigny (Nom patronymique: Cartigny)
- Né le 12 Juin 1975 à Lesquin, nationalité Française
- Marié, 2 enfants
- Maître de Conférences en Informatique à l'Université de Lille1, enseignant à Polytech'Lille
- Recherches effectuées dans l'équipe 2XS du laboratoire LIFL (UMR CNRS 8022)
- Contact :
  - Téléphone: +33 (0)6.81.60.64.83
  - Email: julien.iguchi-cartigny@lifl.fr , julien.iguchi-cartigny@univ-lille1.fr
  - Pages professionnelles: `http://www.lifl.fr/~iguchi-c/`

Fonctions occupées

Maître de conférences **Septembre 2013 –**

LIFL, Polytech'Lille, Université de Lille 1
- Travaux de recherche au Laboratoire LIFL:
  - Sécurité des mécanismes d'hypervision temps-réel
  - Noyaux prouvés légers
- Cours, TD et TP d'Informatique, encadrement de stages, d'apprentis et de projets à Polytech'Lille et l'Université de Lille 1
- Délégation CNRS de Février 2013 à Juin 2013
- Titulaire d'une Prime d'Excellence Scientifique (PES) depuis Janvier 2012

Maître de conférences **Septembre 2005 – Août 2013**

XLIM, Université de Limoges
- Travaux de recherche au Laboratoire XLIM:
  - Septembre 2005 – Février 2008 : sécurité des réseaux ad hoc et anonymat sur VoIP
  - Mars 2008 – Août 2013 : sécurité des systèmes embarqués, équipe SSD dirigée par le professeur Jean-Louis Lanet
- Cours, TD et TP d'Informatique, encadrement de stages et de projets au département d'Informatique
- Congé pour Recherche et Conversion Thématique (CRCT) de Janvier 2011 à Août 2011
- Titulaire d'une Prime d'Excellence Scientifique (PES) depuis Janvier 2012

Chargé de Recherche **Octobre 2004 – Août 2005**

Japan Advanced Institute of Science and Technology (JAIST), Ishikawa, Japon
- Travaux de recherche dans le *Dependable Distributed Systems Lab (DDSG)* du professeur Xavier Défago:
  - protocoles de routage pour réseaux ad hoc
  - organisation distribuée entre robots mobiles.
- Bourse *"Postdoctoral Fellowship for Foreign Researchers"* du *Japan Society for the Promotion of Science (JSPS)* (organisme gouvernemental), dossier pré-sélectionné par le CNRS, dans le cadre des accords *"Application through an Overseas Nominating Authority"*

Attaché Temporaire d'Enseignement et de Recherche (mi-temps) **Octobre 2003 – Août 2004**

LIFL, Université des Sciences et Technologie de Lille 1
- Travaux de recherche sur la diffusion dans les réseaux ad hoc, équipe Recherche et Développement sur le Dossier Portable (RD2P) du Laboratoire d'Informatique Fondamentale de Lille (LIFL)
- Cours, TD et TP d'Informatique, encadrement de stages et de projets pour l'UFR d'IEEA

Vacataire de l'Enseignement Supérieur **Septembre 1999 – Août 2003**

LIFL, Université des Sciences et Technologie de Lille 1
- Travaux de recherche sur la diffusion dans les réseaux ad hoc, équipe Recherche et Développement sur le Dossier Portable (RD2P) du Laboratoire d'Informatique Fondamentale de Lille (LIFL)
- Cours, TD et TP d'Informatique à l'UFR d'IEEA
- TD et TP d'Informatique à l'IUT A d'Informatique de l'Université de Lille 1 et à l'école Télécom Lille 1
- Titulaire d'une Allocation Ministérielle MNERT (Septembre 2000 – Août 2003)
- Titulaire d'une bourse sur critères universitaires durant le DEA (1999)

Formations

Postdoctorat **Octobre 2004 – Juillet 2005**

Japan Advanced Institute of Science and Technologies (JAIST), Ishikawa, Japon

- Recherche sur les protocoles de routage pour réseaux ad hoc et l'organisation distribuée entre robots mobiles sous la direction du professeur Xavier Défago

Doctorat d'informatique **2000 – 2003**

Équipe RD2P, LIFL, Université des Sciences et Technologies de Lille 1

- Sujet : "Contributions à la diffusion dans les réseaux ad hoc"
- Encadrement : professeur Vincent Cordonnier et Docteur David Simplot-Ryl
- Thèse soutenue le 19 Décembre 2003, Mention Très Honorable
- Composition du Jury:
  - Mme. Ana Cavalli, Professeur à l'Institut National des Télécommunications d'Evry (Rapporteur)
  - M. Éric Fleury, Professeur à l'INSA de Lyon (Rapporteur)
  - M. Ivan Stojmenovic, Professeur à l'Université d'Ottawa, Canada (Rapporteur)
  - M. Vincent Cordonnier, Professeur à l'Université de Lille 1 (Examinateur)
  - M. Piet Demeester, Professeur à l'Université de Gand, Belgique (Examinateur)
  - M. Jean-Marc Geib, Professeur à l'Université de Lille 1 (Examinateur)
  - M. David Simplot-Ryl, Maître de Conférences à l'Université de Lille 1 (Examinateur)

DEA d'Informatique **1999 – 2000**

UFR IEEA, Université de Lille 1

- Mention Bien
- Stage réalisé dans l'équipe RD2P du LIFL, sur le thème des réseaux ad hoc pour étiquettes électroniques

Diplôme de Maîtrise d'informatique **1998 – 1999**

UFR IEEA, Université de Lille 1

- Mention Assez Bien
- Séjour ERASMUS à l'Université de Manchester, UK (Janvier – Août 1999)

Diplôme de Licence d'informatique **1997 – 1998**

UFR IEEA, Université de Lille 1

- Mention Assez Bien

Diplôme de DEUG Science Mathématiques et Informatique Appliqués aux Sciences **1995 – 1997**

UFR IEEA, Université de Lille 1

Participation à des jurys de thèse

Céline Burgod — **Octobre 2009**

Contribution à la sécurité des réseaux ad hoc — *Limoges*

Composition du Jury:

- Mme. Maryline Laurent-Maknavicius, professeur, TELECOM SudParis (Rapporteur)
- M. Ludovic Mé, professeur à Supélec Rennes (Rapporteur)
- M. Carlos Aguilar Melchor, Maître de conférences à l'Université de Limoges (Examinateur)
- M. Christophe Bidan, Maître de Conférences à Supélec Rennes (Examinateur)
- M. Jean-Pierre Borel, professeur des Universités à l'Université de Limoges (Examinateur)
- M. Yves Deswarte, Directeur de Recherche CNRS au LAAS-CNRS (Examinateur, Président)
- M. Julien Iguchi-Cartigny, Maître de Conférences à l'Université de Limoges (Examinateur)

Ahmadou Al Khary Séré — **Septembre 2010**

Tissage de contre-mesures pour machine virtuelle embarquée — *Limoges*

Composition du Jury:

- Mme. Assia Tria, professeur à EMSE Gardannes (Rapporteur)
- Mme. Isabelle Simplot-Ryl, professeur à Université de Lille 1 (Rapporteur)
- M. Pierre Girard, Ingénieur de recherche Gemalto (Examinateur)
- M. Aurélien Francillon, PhD ETH Zurich (Examinateur)
- M. Philippe Gaborit, professeur à l'Université de Limoges (Examinateur, Président)
- M. Jean-Louis Lanet, professeur à l'Université de Limoges (Examinateur)
- M. Julien Iguchi-Cartigny, Maître de Conférences à l'Université de Limoges (Examinateur)

Agnès Noubissi — **Septembre 2011**

Mise à jour dynamique pour cartes à puce Java — *Limoges*

Composition du Jury:

- M. Daniel Hagimont, professeur à l'INPT/ENSEEIHT (Rapporteur, Président)
- M. Gilles Grimaud, professeur à l'Université de Lille (Rapporteur)
- M. Pierre Girard, Ingénieur de recherche Gemalto (Examinateur)
- M. Jean-Louis Lanet, professeur à l'Université de Limoges (Examinateur)
- M. Julien Iguchi-Cartigny, Maître de Conférences à l'Université de Limoges (Examinateur)

Nassima Kamel — **Décembre 2012**

Détection d'attaques web sur serveur web embarqué — *Limoges*

Composition du Jury:

- Mme. Samia Bouzefrane, professeur au CNAM (Rapporteur)
- Mme. Marie-Laure Potet, professeur à l'ENSIMAG (Rapporteur, Président)
- M. Christophe Bidan, Maître de Conférences à Supélec Rennes (Examinateur)
- M. Christophe Clavier, professeur à l'Université de Limoges (Examinateur)
- M. Jean-Louis Lanet, professeur à l'Université de Limoges (Examinateur)
- M. Julien Iguchi-Cartigny, Maître de Conférences à l'Université de Limoges (Examinateur)

Organisation de conférences

- Participation à l'organisation des Journées Cryptographie et Sécurité de l'Information à Limoges en 2006, 2008 et 2011.

Rayonnement

- Co-éditeur du LNCS 6035, Smart Card Research and Advanced Applications (9th IFIP WG 8.8/11.2) International Conference, CARDIS 2010, Passau, Germany, Avril 14-16, 2010. Sélection de papiers publiée dans Lecture Notes in Computer Science Vol. 6035 Springer 2010.
- Membre du comité de programme des conférences :
  - 4ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information (SARSSI 2009), Luchon, France, 22 - 26 juin, 2009
  - 7th International Conference on Risks and Security of Internet and Systems (CRiSIS 2012), Cork, Ireland, October 10-12, 2012
  - International Workshop on Information Security, Theory and Practice (ISTP-2012) December 11-12, 2012, London, UK.
- Expert auprès de la DGA pour l'évaluation de projets d'études dans le cadre de la procédure "Recherche exploratoire" (2009)
- Expert auprès de l'ANR pour le programme INS 2011 et 2012

Enseignement

Liste des enseignements depuis 2005.....

Note la répartition cours, Travaux Dirigés (TD) et Travaux Pratiques (TP) selon les matières et les années se trouve en annexe page 6.

Programmation Internet (Prog. Internet)
Développement HTML, CSS et JavaScript — *Licence 2 d'Informatique*
- Enseignant TD et TP

Réseaux 1 (RSX1)
Introduction aux réseaux — *Licence 2 d'Informatique*
- Enseignant travaux pratiques

Base de données 2 (BD2)
Développement PHP, sécurisation des applications webs — *Licence 3 d'Informatique*
- Enseignant cours, TD et TP
- Création d'un cours et de travaux pratiques sur le langage PHP et sur la sécurité des applications webs

Réseaux2 (RSX2)
Configuration d'un réseau local: Ethernet, IP, firewall, VLAN, STPLicence 3 d'Informatique & GIS 4ème Apprentissage
- Enseignant cours, TD et TP
- Création d'un cours et de travaux pratiques sur le déploiement d'un réseau local à l'aide du logiciel Netkit

Java
Développement Java: langage, API, tests unitaires — *Licence 3 d'Informatique*
- Enseignant cours et TP
- Création de plusieurs cours sur les collections, les entrées-sorties et les tests unitaires

Projets L3
Encadrement de projets en Licence 3 (voir page 5) — *Licence 3 d'Informatique*

Développement d'applications distribuées (DAD)
Développement J2EE / JEE — *Master 1 d'Informatique & GIS 4ème Apprentissage*
- Enseignant cours et TP
- Développement de cours et de travaux pratiques sur:
  - développement J2EE/JEE avec les technologies Spring, JPA, spring data JPA et Spring MVC
  - mise au point de tests unitaires et d'intégration
  - évaluation de la qualité de code avec des outils de métriques (Sonar)
  - développement collaboratif à l'aide des systèmes de gestion de sources distribués (Mercurial)

Projets M1
Encadrement de projet en Master 1 (voir page 5) — *Master 1 d'Informatique*

Admin. et Sécurité des Syst. d'exploitation et des réseaux (ASSR)
Sécurité offensive et défensive des aspects systèmes et réseaux — *Master 2 CRYPTIS, parcours sécurité*
- Enseignant cours et TP
- Développement de cours sur
  - sécurité des réseaux, pentesting, sécurité wifi,
  - recherche et exploitation de vulnérabilités web
  - renforcement de la sécurité des systèmes d'exploitation, travaux pratiques avec Linux
  - politiques de sécurité
  - virtualisation des systèmes d'exploitation
- Accueil d'intervenants pour les parties spécialisées dans le *reverse-engeenering*, l'audit web, l'écriture d'exploits systèmes et la détection d'intrusion (IDS)

Système d'exploitation
Introduction aux systèmes d'exploitation — *Master 2 CRYPTIS, parcours sécurité*
- Enseignant cours et TP
- Création d'un cours sur les mécanismes internes de sécurité d'un système d'exploitation: isolation spatiale et temporelle, renforcement du système, mécanismes dédiés à la sécurité dans Linux

Programmation orientée Objet (Prog. Objet)
Programmation Java pour un public de mathématiciens — *Master 2 CRYPTIS, parcours cryptographie*
- Enseignant cours et TP
- Création d'un cours de Java (collections, entrées-sorties, API Cryptographie) à destination d'un public de mathématiciens afin de faciliter la transition vers le développement Java Card

Cryptographie et Applications (Crypto.

Programmation sécurisée à l'aide d'API cryptographiques *Master 2 CRYPTIS, parcours commun*

- Enseignant cours et TP
- Création d'un cours sur le développement OpenSSL et GnuTLS à destination d'un double public informaticien et mathématicien

Encadrement de projets

Encadrement depuis 2005 de plus 60 projets d'étudiants de Licence 3 (module Projet L3), Master 1 (module projet M1) et Master 2 (projet avancé dans le module ASSR). La liste qui suit est un sous-ensemble significatif :

- Réseaux:
  - Maquette Kerberos, LDAP et NFS (2008)
  - Maquette CISCO sous GNS3 (2010)
  - Maquette IPv6 sous Netkit (2010)
  - Maquette MPLS sous GNS3 (2010)
  - Maquette DNSSEC sous Netkit (2012)
- Sécurité réseaux:
  - Fuzzing HTTP (2011)
  - Outil d'évaluation de la sécurité d'une installation SPIP (2012)
  - Audit de sécurité du système d'information de partenaires institutionnels et privés (2009 – 2013)
  - Corruption et détournement de trafic réseau sur les protocoles Ethernet, STP et VLAN (2012)
  - Audit de la sécurité d'une installation WIFI (2013 – 2014)
  - Audit de la surface d'attaque d'un SI (2013 – 2014)
- Sécurité système:
  - Étude du mécanisme de permissions Android (2012)
  - Exploitation système par *buffer overflow* et *integer overflow* (2009 – 2010)
  - Écriture d'une *rootkit* Linux (2008)
  - Écriture d'un logiciel de *backdoor* pour Windows (2009)
  - Étude de la similarité de binaires (2012 – 2013)
  - Architecture de sécurité web (2013 – 2014)
- Développement embarqué:
  - Développement d'un OS *barebone* minimaliste sur AT91 (2012)
  - Portage du serveur web SMEWS sur AT91 (2010)
  - Support *sniffing* pour Netkit (2009)

Administration de l'enseignement

- Membre du jury de la Licence 3 d'Informatique, du Master 1 d'informatique, et du Master 2 CRYPTIS
- Organisation de séance de relecture CV et lettre de motivation dans le Master M2 CRYPTIS
- Administration système et réseaux de la salle du M2 CRYPTIS

Collaborations Internationales

- Participation au consortium USTH commun à plusieurs universités françaises pour la création d'un master d'Informatique à l'Université de Hanoï (Vietnam) en Octobre 2012 et porte sur la sécurité des réseaux.

Répartition des enseignements

	2005-2006	2006-2007	2007-2008	2008-2009	2009-2010	2010-2011	2011-2012	2012-2013	2013-2014
Prog. Internet (L2)	0/64/0	0/64/0							
RSX1 (L2)	0/0/15	0/0/15							
BD2 (L3)		9/0/0	9/0/0	9/0/0	9/0/0	9/0/0	9/0/0	9/3/36	
RSX2 (L3 + GIS4A)	9/27/30	9/27/30	9/15/22.5	9/15/22.5	9/0/12	9/0/0	9/0/0	9/0/0	18/0/12
Java (L3)			4,5/14/0		3/0/21	3/0/0	3/0/0	3/0/0	
Projets (L3)	0/0/32	0/0/32	0/0/20	0/0/94	0/0/55	0/23/0	0/12/0	0/8/0	
DAD (M1 + GIS4A)	7/30/30	7/30/30	21/9/60	21/0/33	21/0/39	21/0/0	24/0/108	21/0/78	44/0/36
Projets (M1)	0/0/16	0/0/16		0/0/48	0/0/72	0/17/0	0/9/0	0/8/0	0/22/0
Prog. Objet (M2)			9/9/0	8/0/8	7.5/7.5/0	8/7/0	8/7/0	8/7/0	
ASSR (M2)			6/19/0	27/22/16	10.5/10.5/15	16.5/16.5/15	25/25/15	25/25/15	
SE (M2)	14/17/0	14/17/0	9/9/0						
Crypto. (M2)	0/0/12	0/0/12	0/0/18						
Réseaux (Vietnam)								0/30/0	
Total	30/138/135	30/138/135	67.5/75/120,5	74/37/221,5	60/18/214	66.5/63.5/15	78/53/123	75/81/129	124h
Équ. heures TD	273h	273h	257h	296h	251h	174h (CRCT)	252h	280h	

Annexe B

Sélection d'articles

[CIC3] Guillaume Bouffard, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Combined Software and Hardware Attacks on the Java Card Control Flow ». In : *Smart Card Research and Advanced Applications*. Sous la dir. d'Emmanuel Prouff. T. 7079. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, p. 283–296. isbn : 978-3-642-27256-1. doi : 10.1007/978-3-642-27257-8_18.

[CIC4] Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet et Julien Cartigny. « SmartCM : A Smart Card Fault Injection Simulator ». In : *Proc. IEEE International Workshop on Information Forensics and Security (WIFS 2011)*. Foz do Iguaçu, Brazil : IEEE, nov. 2011, p. 1–6. isbn : 978-1-4577-1019-3, 978-1-4577-1017-9, 978-1-4577-1018-6. doi : 10.1109/WIFS.2011.6123124. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6123124.

[WI2] Agnes C. Noubissi, Julien Iguchi-Cartigny et Jean-Louis Lanet. « Hot updates for Java based smart cards ». In : *Proc. of the Third Workshop on Hot Topics in Software Upgrades (HotSwapUp'11)*. IEEE, avr. 2011, p. 168–173. isbn : 978-1-4244-9195-7. doi : 10.1109/ICDEW.2011.5767630. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5767630.

[JIC1] Ahmadou Séré, Jean-Louis Lanet et Julien Iguchi-Cartigny. « Evaluation of Countermeasures Against Fault Attacks on Smart Cards ». In : *International journal of Security and Its Applications* 5.2 (2011), p. 49–60.

[JIC2] Julien Iguchi-Cartigny et Jean-Louis Lanet. « Developing a Trojan applets in a smart card ». In : *Journal in Computer Virology* 6.4 (nov. 2010), p. 343–351. issn : 1772-9890, 1772-9904. doi : 10.1007/s11416-009-0135-3. url : http://link.springer.com/10.1007/s11416-009-0135-3.

Combined Software and Hardware Attacks on the Java Card Control Flow

Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet

Smart Secure Devices (SSD) Team – XLIM Labs, Université de Limoges
83 Rue d'Isle, 87000 Limoges, France
{guillaume.bouffard,julien.cartigny,jean-louis.lanet}@xlim.fr

Abstract. The Java Card uses two components to ensure the security of its model. On the one hand, the byte code verifier (BCV) checks, during an applet installation, if the Java Card security model is ensured. This mechanism may not be present in the card. On the other hand, the firewall dynamically checks if there is no illegal access. This paper describes two attacks to modify the Java Card control flow and to execute our own malicious byte code. In the first attack, we use a card without embedded security verifier and we show how it is simple to change the return address of a current function. In the second attack, we consider the hypothesis that the card embeds a partial implementation of a BCV. With the help of a laser beam, we are able to change the execution flow.

Keywords: Java Card, control flow, laser, Java Card Stack, attack

1 Introduction

Java Card is a kind of smart card that implements one of the two editions, "*Classic Edition*" or "*Connected Edition*", of the standard Java Card 3.0 [8]. Such smart cards embed a virtual machine (VM) which interprets codes already romized with the operating system or downloaded after issuance [1]. In fact, Java Card is an open platform for smart cards, *i.e.* able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, byte codes delivered by the Java compiler and the converter (in charge of giving a compact representation of class files) are safe, *i.e.* the loaded application is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks application permissions and access in the card, enforcing isolation between them.

Java Cards have shown improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. Currently, the most powerful attacks are

[1] Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [3]. This protocol ensures that the owner of the code has the necessary authorization to perform the action.

hardware based attacks and particularly fault attacks. A fault attack modifies parts of memory content or a signal on internal bus, which can lead to deviant behavior exploitable by an attacker. A comprehensive consequence of such attacks can be found in [6]. Although fault attacks have been mainly used in the literature from a cryptanalytic point of view (see [1,4,9]), they can be applied to every code layers embedded in a device. For instance, while choosing the exact byte of a program the attacker can bypass counter-measures or logical tests. We called *mutant* such modified application.

2 State of the Art

2.1 Java Card Security

The Java Card platform is a multi-application environment where critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers to create private namespaces for applets. In a smart card, complying with the traditional enforcement process is not possible. On the one hand, the type verification is executed outside the card due to memory constraints. On the other hand, the class loader and security managers are replaced by the Java Card firewall.

To allow code to be loaded into the card after post-issuance raises the same issues as the web applets. An applet not built by a compiler (handmade byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. Java is a strongly typed language where each variable and expression has a type determined at compile-time, so that if a type mismatches from the source code, an error is thrown. The Java byte code is also typed. Moreover, local and stack variables of the VM have fixed types even in the scope of a method execution but no type mismatches are detected at run time, and it is possible to make malicious applets exploiting this issue. For example, pointers are not supported by he Java programming language although they are extensively used by the Java VM (JVM) where object references from the source code are relative to a pointer. Thus, the absence of pointers reduces the number of programming errors. But it does not stop attempts to break security protections with unfair uses of pointers.

The BCV is an essential security component in the Java sandbox model: byte code alteration contained in an ill-typed applet may induce a security flaw. The byte code verification is a complex process involving elaborate program analyses using a very costly algorithm in time consumption and memory usage. For these reasons, lot of cards do not implement this kind of component and rely on the responsibility of the organization which signs the code of the applet to ensure that they are well-typed.

The separation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of context. When an applet is created, the Java Card Runtime Environment (JCRE) uses an unique Applet IDentifier (AID) whose it is possible to retrieve the name of the package which defines it. If two applets are an instance of classes of the same Java Card package, they are considered in the same context. There is also a super user context, called the JCRE context. Applets associated with this context can access to objects from any other context on the card.

Each object is assigned to an unique owner context which is the context of the created applet. An object method is executed in the owner object context. This context provides information allowing, or not, to access to another object. The firewall prevent a method executing in one context from access to any attribute or method of objects to another context.

2.2 The CAP File

The CAP (for Convert APplet) file format is based on the notion of components. It is specified by Oracle [8] as consisting of ten standard components: `Header`, `Directory`, `Import`, `Applet`, `Class`, `Method`, `Static Field`, `Export`, `Constant Pool` and `Reference Location` and one optional: `Descriptor`. Moreover, the targeted Java Card VM (JCVM) may support user `custom` components. We except the `Debug` component because it is only used on the debugging step and it is not sent to the card.

Each component has a dedicated role and it has linked between them. A modification, voluntary or not, of a component is difficult and may provide unmeaning file. An invalid file is often detected during the installation step by the target JCVM.

2.3 Logical Attacks

The Hubbers and Poll's Attack Erik Hubbers *et al.* made a presentation at CARDIS 2008 about attacks on smart card. In their paper [5], they present a quick overview of the classical attacks available and gave some counter-measures. They described four methods:

1. CAP file manipulation,
2. Fault injection,
3. Shareable interfaces mechanisms abuse and
4. Transaction Mechanisms abuse

The goal of (1) is to modify the CAP file after the building step to bypass the BCV. The problem is that, like explained before, an on-card BCV is an efficient system to block this attack. Using the fault injection in (2), the authors succeed to bypass the BCV. Even if there is not particular physical protection,

this attack is efficient but quiet difficult to perform and expensive.

The idea of (3) to abuse shareable interfaces is really interesting and can lead to trick the VM. The main goal is to obtain a type confusion without the need to modify the CAP files. To do that, the authors create two applets which communicate using the shareable interface mechanism. To create a type confusion, each applet use a different type of array to exchange data. During compilation or on loading, there is no way for the BCV to detect a problem. But it seems that every card tried, with an on-card BCV, refused to allow applets using shareable interface. As it is impossible for an on-card BCV to detect this kind of anomaly, Hubbers *et al.* emitted the hypothesis that any use of shareable interface on card can be forbidden with an on-board BCV.

The last option left is the transaction mechanism (4). The purpose of transaction is to make a group of atomic operations. Of course, it is a widely used concept, for instance in databases, but still complex to implement. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction and reset references to such objects to null. However, Hubbers *et al.* found some cases where the card keeps the reference to objects allocated during transaction even after a rollback.

Moreover, the authors described the easiest way to make and exploit a type confusion to gain illegal access to otherwise protected memory. A first example is to get two arrays with different types, a byte and a short array. If a byte array of 10 bytes is declared and it exists a reference to a short array, it is possible to read 10 shorts, so 20 bytes. With this method they can read the 10 bytes stored after the array. If Hubbers *et al.* increase the size of the array, they will able to read as much memory as they want. The main problem is more *how to read memory before the array?*

The other used confusion is between an array of bytes and an object. If Hubbers *et al.* put a byte as first object attribute, it is bound to the array length. Then it is really easy to change the length of the array using the reference to the object. With this attack, the problem becomes *how to give a reference to an object for another object type?*

Barbu *et al.*'s Attack: Combined Physical & Logical Attack At CARDIS 2010, Barbu *et al.* described a new kind of attack in their paper [2]. This attack is based on the use of a laser beam which modifies a runtime type check (the `checkcast` instruction) while running. This applet was checked by the on-card BCV, considered as valid, and installed on the card. The goal is to cause a type confusion to forge a reference of an object and its content. We consider three classes `A`, `B` and `C`. They are declared in the listing 1.1.

```
public class A {
   byte b00, ..., bFF
}
```

```
public class B {
   short addr
}
```

```
public class C {
   A a;
}
```

Listing 1.1. Classes used to create a type confusion.

The cast mechanism is explained in the JCRE specification [8]. When casting an object to another, the JCRE dynamically verifies if both types are compatible, with a `checkcast` instruction. Moreover, an object reference depends of the card architecture. The following example can be used:

```
T1 t1;                 aload @t1
T2 t2 = (T2) t1; <=>   checkcast T2
                       astore @t2
```

The authors want cast an object `b` to an object `c`. If `b.addr` is modified to a specific value, and if this object is cast to a `C` instance, you may change the referenced address by `c.a`. But the `checkcast` instruction prevents from this illegal cast.

Barbu *et al.* use in his `AttackExtApp` applet (listing 1.2) an illegal cast at line 9.

```
1  public class AttackExtApp extends Applet {
2      B b; C c; boolean classFound;
3      ... // Constructor, install method
4      public void process(APDU apdu) {
5        ...
6        switch (buffer[ISO7816.OFFSET_INS]) {
7          case INS_ILLEGAL_CAST:
8            try {
9              c = (C) ( (Object) b );
10             return; // Success, return SW 0x9000
11           } catch (ClassCastException e) {
12             /* Failure, return SW 0x6F00 */
13           }
14           ... // more later defined instructions
15 }    } }
```

Listing 1.2. `checkcast` type confusion

This cast instruction throws a `ClassCastException` exception. With specific material (oscilloscope, *etc.*), the thrown exception is visible in the consumption

curves. With a time-precision attack, the authors prevent with an injection by a laser based fault the `checkcast` to be thrown. When the cast is done, the references of `c.a` and `b.addr` link the same value. Thus, the `c.a` reference may be changed dynamically by `b.addr`. This trick offers a read/write access on smart card memory within the fake `A` reference. Thanks to this kind of attack, Barbu *et al.* can apply their combined attack to inject ill-formed code and modify any application on Java Card 3.0, such as EMAN1 [6].

3 EMAN2: A Stack Underflow in the Java Card

3.1 Genesis

The aim of this attack is to modify the register which contains the method return address by the address of an array which contains our malicious byte code. To succeed, the target smart card has no BCV and we know its loading keys. For this work, we have used two tools developed in the Java-language. The first one, the CFM [11] (for CAP File Manipulator) provides a friendly way to parse and full-modify the CAP files. The other one is the Java library OPAL [10] used to communicate with the card. So, to perform this attack, we must:

1. find the array address which contains the malicious byte code;
2. find where is located, in the Java Card stack, the address of the return function;
3. change this address by the address of the byte codes contained in our malicious array.

We will explain each step in the next subsections.

3.2 How to obtain the Address of our Malicious Array?

In a previous work [6], we explained how to execute auto-modifiable code in a Java Card. This malicious byte code was stored in an byte-array and called by ill-formed applet. We also have to remember the way to obtain the array address.

```
1 public short getMyAddressByteArray (byte[] array) {
2   short foo=(byte)0x55AA;
3   array[0] = (byte)0xFF;
4   return foo;
5 }
```

Listing 1.3. method to retrieve the address of an array

In order to retrieve the address of an array, we have implemented the method `getMyAddressByteArray` described in the listing 1.3. In its unmodified version, it returns the value contained in `foo`. The instruction in line 3 uses an array given in the function parameter. As seen in listing 1.4, the JCVM needs first to push a reference to the array `tab`[2]. Finally the function returns the previously

[2] In our tested card, all references are performed in a `short` type

pushed value of `foo`.

If an event changed our byte code like described in the listing 1.5, our function directly returns the reference of the array given as parameter. To make this modification, we use the CFM to "`nop`" each instruction between *push the array reference* and *return the short pushed value*. These instructions are written in a bold font in the listing 1.5. Using a card without BCV, a applet containing this function provides address of each array given in its parameter. The returned address is locate in the EEPROM area.

```
public short
getMyAddressByteArray
                (byte[] array) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
03          sconst_0
02          sconst_m1
39          sastore
1E          sload_2
78          sreturn
}
```

Listing 1.4. The Java byte code corresponding to the function 1.3

```
public short
getMyAddressByteArray
                (byte[] array) {
03 // flags: 0 max_stack : 3
21 // nargs: 2 max_locals: 1
10 AA       bspush      -86
31          sstore_2
19          aload_1
00          nop
00          nop
00          nop
00          nop
78          sreturn
}
```

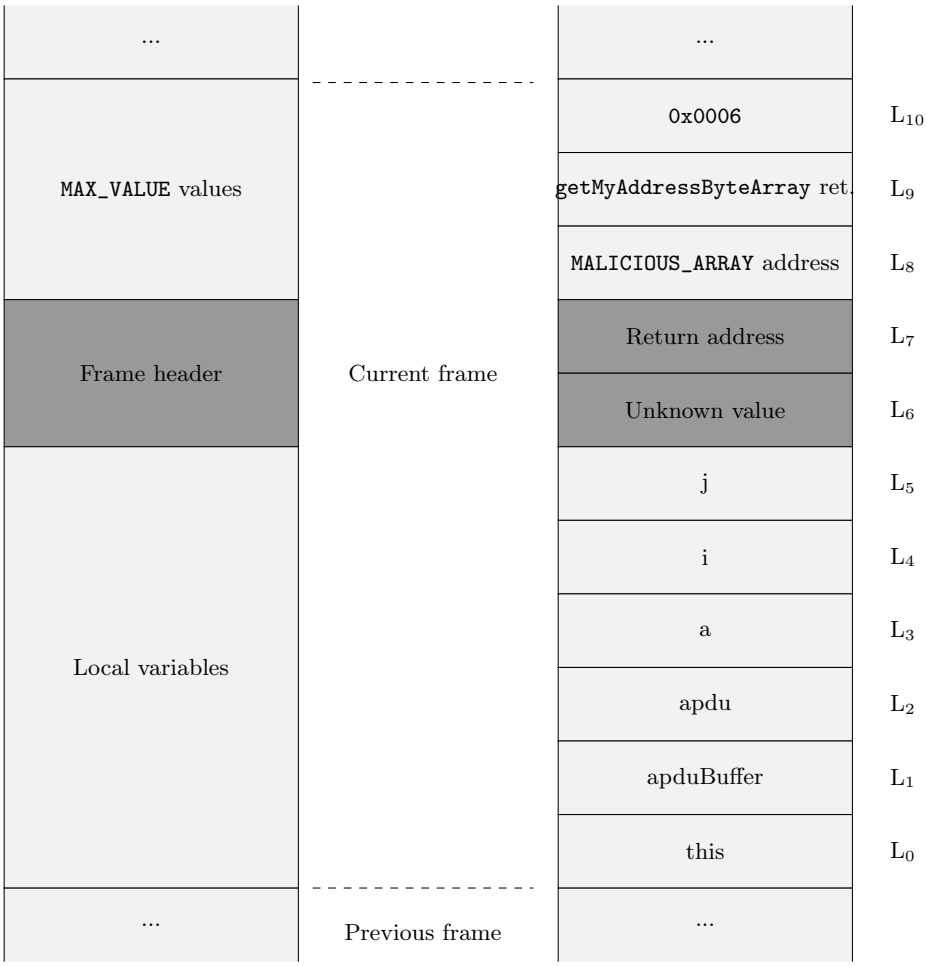
Listing 1.5. The function 1.3 with the modified return

On the targeted JCVM, the returned address by the malicious function `getMyAddressByteArray` does not refer to the array data. It is a pointer on the array header which is structured by 6 bytes that include the type and the number of contained elements. if the array is transient, the RAM array address follows the header. Else, the array data is stored after the 6-byte header.

3.3 Java Card Stack

To perform this attack, we should understand the Java Card stack. In fact, a Java Card contains two stacks, the native and the JCVM stack. The first one is used by the smart card operating system. The second one, is used by the JCVM to execute some Java applets value pushed in the Java Card stack.

To characterize the Java Card stack, We used the method `ModifyStack`, listed in 1.6. This method has three parameters: `apduBuffer`, a reference to a byte array; `apdu`, a reference to an instance of the `APDU` class; and `a`, a short value. The figure 1(b) represents the Java Card stack where each method parameter, variable and a reference to the class instance (`this`) are stored in the local variables area. Next, the information present in the frame header (in L_6 and L_7) are important data which hold the method return address. Finally, the



(a) Generic Java Card stack

(b) An Example of the Java Card Stack

Fig. 1. Java Card stack characterization

stack contains the data pushed while the method runs[3].

The BCV must checks several points. In particular: it should prevent any violations of the memory management (illegal reference access), stack underflow or overflow. This means these checks are potentially not verified during runtime and thus can lead to vulnerabilities. The Java frame is a non persistent data

[3] The maximum number of values to push is defined in the field `MAX_STACK` included in each Java Card method header.

```
1 public void ModifyStack(byte[] apduBuffer, APDU apdu, short a)
2 {
3   short i=(short) 0xCAFE;
4   short j=(short) (getMyAddressByteArray(MALICIOUS_ARRAY)+6);
5   i = j ;
6 }
```

Listing 1.6. Function to modify the Java Card stack

structure implemented in different manners and the specification gives no design direction for it.

3.4 Our Attack

Our attack aims to change the index of a local variable[4]. We purpose to use two instructions: `sload` and `sstore`. As described in the JCVM specification [8], these instructions are normally used in order to load a short value from a local variable and to store a short value in a local variable. The CFM allows us to modify the CAP file in order to access the system data and the previous frame. As example, the code in the listing 1.6, line 4, stores the value returned by `getMyAdressByteArray()` and adds 6 into variable `j`. Then, it loads the value of `j`, and stores it into variable `i` (line 5).

So, if we change the operand of `sload` (`sload 5`, at the offset `0x0F` of the listing 1.7) we store information from a non-authorized area into the local 5. Then, this information is sent out using an APDU. We tried this attack using a +2 offset and we retrieved the short value `0x8AFA` which was the address of the caller. Thus, we able to read without difficulty in the stack after our local variables. Furthermore, we can write anywhere into the stack below: there is no counter-measures. The targeted smart card implements an interpreter that relies entirely on the byte code verification process.

Next, we modified the CAP file to change the return address by our malicious array address, this step was explained in the section 3.2. When this modification is performed, the exception `0x1712` will be throw. So, we proved within this applet that we can redirect the control flow of such a JCVM.

3.5 Counter-measure

As we said, no important knowledge are needed in Java Card security and the simple modifications of an CAP file, with the tool [11], may perform these at-

[4] The specification says that the maximum number of variables that may be used in a method is 255. It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked.

```
      public void ModifyStack
             (byte[] apduBuffer, APDU apdu, short a) {
0x00:    02 // flags: 0   max_stack:  2
0x01:    42 // nargs: 4   max_locals: 2
0x02:    11 CA FE   sspush          0xCAFE
0x05:    29 04      sstore          4
0x07:    18         aload_0
0x08:    7B 00      getstatic_a     0
0x0A:    8B 01      invokevirtual   1
0x0C:    10 06      bspush          6
0x0E:    41         sadd
0x0F:    29 05      sstore          5
0x11:    16 05      sload           5
0x13:    29 04      sstore          4
0x15:    7A         return
      }
```

Listing 1.7. Malicious byte code applet of the function 1.6

tacks.

The principle of the stack underflow is to get access to memory area normally used by the system to the previous frame. A simple counter-measure would consist in checking the number of locals and arguments provided in the header of the method. With this simple check one cannot gain access to the system area where the JPC (previous Java Program Counter) and SPC (previous Stack Pointer) are stored. In order to avoid parsing the previous frame; the implementation can use the linked frame approach like in the simple RTJ VM references. This approach implies to create a new frame, to copy the argument of the current frame into the new, instead of the implemented method which uses the current stack as the beginning of the new frame. Desynchronizing the frames will avoid simply a stack underflow attack.

4 EMAN4: Modifying the Execution Flow with a Laser Beam

4.1 Description of our attack

In the section 3, we supposed there is no BCV. This hypothesis allowed us to modify the CAP file before loading it on the card. For the following, the targeted card has an improved security system based on a partial implementation of a BCV. This component statically checks the byte codes during the loading step and dynamic byte code checks are done during the runtime.

To perform this attack, we provide an external modification, such as the Barbu *et al.*'s attack, with a laser beam to change the control flow to execute

our own malicious byte code. Furthermore, we have the smart card loading keys.

In order to modify the execution flow, we will use the `for` loop properties. Next, after having understood how this kind of loop works, we modify it to change the control flow.

4.2 How Re-loop a For Loop

The `for` loop is probably the most widely loop used in the imperative programming languages. A classic `for` loop, such as in the listing 1.8, may be split in three parts. The first one is the declaration of the loop with the preamble (the initialization of the loop), followed by the stop condition and a function executed at each iteration. Next, the loop body contains the executed instructions for each iteration. Finally, a jump-like instruction re-loop to the next iteration if the stop condition is not satisfied.

```
for (short i=0 ; i<n ; ++i){
 foo = (byte) 0xBA;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions have
 // been hidden for a
 // better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

Listing 1.8. A `for` loop

```
0x00: sconst_0
0x01: sstore_1
0x02: sload_1
0x03: sconst_1
0x04: if_scmpge_w       00 7C
0x07: aload_0
0x08: bspush            BA
0x0A: putfield_b        0
0x0C: aload_0
0x0D: getfield_b_this   0
0x0F: putfield_b        1
// Few instructions have
// been hidden for a
// better meaning.
0xE3: aload_0
0xE4: getfield_b_this   1
0xE6: putfield_b        0
0xE8: sinc              1 1
0xEB: goto_w            FF17
```

Listing 1.9. Associated byte codes of the loop 1.8

According to the amount of instructions contained in the loop body, the re-loop instruction has relative offset on 1 or 2-byte (± 127 or ± 255 bytes). In the Java Card byte code, the re-loop instruction may be a `goto` or `goto_w`. For our attack, we are interested in the `goto_w` statement at the offset `0xEB` (listing 1.9).

4.3 Our Attack:

To begin, we install into a Java Card an applet which contains the loop `for` described in the listing 1.8. The function which contains this loop is compliant

with each security rule of Java Card and the embedded smart card BCV allows its loading.

An external modification based on a laser beam against the `goto_w` instruction, at the offset `0xEB` in the listing 1.9, may change the control flow of the applet. We would like to redirect this flow in the array `MALICIOUS_ARRAY` to execute our malicious byte code. Thus, changing the `goto_w` parameter `0xFF17` to `0x0017` involves a relative jump to the 17^{th} byte after this instruction. To success attack, our array must locate after the modified function in the EEPROM area.

Smart Card memory management The main difficulty regarding this attack is the memory management. Indeed, the static array `MALICIOUS_ARRAY` must physically put after our malicious function. For that, we analyzed how our targeted smart card stores its data. In order to understand the algorithm used by the card to organize its memory, we did the following method:

1. first, few chosen applets are installed on the card within a careful dump of the EEPROM memory between each install,
2. next, the card is stressed by installing and deleting different applets size. A dump is done at each step.

For each analyzed smart card, we have obtained the same algorithm used to manage the memory. These Java Cards have a *first fit* algorithm where the applet data are stored after its byte code. If the smart card have managed few applets without causing fragmentation, it is likely than the applet data is stored before the corresponding applet byte code.

In our case, when our applet is firstly installed on the card the dump obtained is listed in 1.10.

```
0x0A7F0:  18AE 0188 0018 AE00 8801 18AE 0188 0018
0x0A800:  AE00 8801 18AE 0188 0018 ae00 8801 18ae
0x0A810: 0188 0059 0101 A8FF 177A 008A 43C0 6C88
0x0A820:  abcd ef00 0000 0000 0000 0000 0000 0000
0x0A830:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A840:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A850:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A860:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A870:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A880:  0000 0000 0000 0000 0000 0000 0000 0000
0x0A890:  0000 0000 0000 0000 1117 1200 0000 8D6F
0x0A8A0:  C000 0000 0000 00FE DCBA
```

Listing 1.10. Memory organization of our installed applet

As may be seen in the dump 1.10, the function to fault precedes the array `MALICIOUS_ARRAY` in light-gray. This dump is a linked byte code on the contrary of the byte code listed in 1.9.

The Goto redirection Before injecting our fault, the function returns `0x9000` (status without error).

After precisely targeted the high-byte parameter of the `goto_w` instruction located at `0xA817` in the listing 1.10, a laser beam attack swaps `0xFF17` to `0x0017`. This fault allows to redirect the execution flow. Indeed, the `goto_w` jumps forward to go in the array `MALICIOUS_ARRAY`. A land up area of `nop` catches up the instruction pointer which will execute our malicious code, here an exception thrown the value `0x1712`. This result proves that we succeed to change the control flow of our applet.

Moreover, event if the memory is encrypted, this kind of attack have fifty percent to change the `goto_w` instruction statement to redirect towards the front.

4.4 Counter-measures

Create a mutant application is the same way that changing an applet after its loading. To protect the JCVM against this attack, voluntary or not, we have developed some counter-measures described in [7]. We are going to present a brief resume of these counter-measures.

The XOR Detection Mechanism This protection is based on basic blocks. It allows code integrity and application control flow checking. A basic block is a sequence of instructions with a single entry point and a single exit point[5]. For each basic bloc, a checksum is computed by using the XOR operation on all the bytes composing a basic block. Then this table is stored in the CAP file as a Java Card custom component. The interpreter has to be modified to exploit and verify the checksum information. During runtime, the interpreter computes again the checksum and compares it with the stored values.

The Field of Bit Detection Mechanism This counter-measure checks the nature of the element stored in the byte array of the CAP file. A tag (bit) is associated to each byte of the bytecode. The tag has the value 0 if the bytecode is an opcode, and it has the value 1 if the byte code is a value (a parameter of an *opcode*). During an attack, the following situations can appear:

1. An increase of operands number for the instruction, it is the case when `add` (no operand) is replaced by `icmpeq` (one operand).

[5] The execution of a basic block starts only at an entry point, and leaves a basic block only at an exit point.

2. A decrease of operands number for the instruction, it is the case when `aload` (one operand) is replaced by `athrow` (no operand).
3. No change on operand number: it is the case when an `iload` (one operand) is replaced by a `return` (one operand).

This method can detect when the changing `1` and `2` happen. During the compilation, a field of bit is generated representing the type of each element contained in the method byte array. It is stored also as a Java Card custom component in the CAP file. The interpreter checks before executing an *opcode* that its byte was scheduled to be executed or not.

The Path Check Mechanism This method computes the control flow graph of the method by extracting the basic blocks from the code. The list of paths from the beginning vertex is computed for each vertex of the control flow graph. This computed paths are encoded using the following convention:

1. Each path begins with the tags `0` and `1` to avoid a physical attack that changes it to `0x00` or to `0xFF`.
2. If the instruction that ends the current block is an unconditional or conditional branch instruction when jumping to the target of this instruction, then the tag `0` is used.
3. If the execution continues at the instruction that immediately follows the final instruction of the current block then the tag `1` is used.

If the final instruction of the current basic block is a switch instruction, the is made by any number of bits that are necessary to encode all the targets. When interpreting the byte code, the VM computes the path followed by the program using the same convention; for example, when jumping to the target of a branch instruction it saves the tag `0`. Then prior to the execution of a basic block, the VM checks that the followed path is an authorized path, *i.e.* a path that belongs to the list of path computed for this basic block. In the case of a loop (backward jump) the interpreter checks the path for the loop, the number of references and the number of values on the operand stack before and after the loop, to be sure that for each round the path remains the same.

5 Conclusion

In this paper we have described two ways to change the execution flow of an application after loading it into a Java Card. The first method, EMAN2, provides a way to change the return address of the current method contained in its frame stack. This attack is possible because there is no check during the stack operations. The second method, EMAN4, uses a laser beam to modifies a well-formed applet loaded and installed on the card to become mutant, even with the on-board BCV.

These two attacks allow to execute malicious code in the JCVM without to be detected by the firewall component. In the case of EMAN2, we have proposed two counter-measures. To opposite, EMAN4 needs a good knowledge of the targeted JCVM and to find the faulted area with the laser beam.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.: Fault attacks on RSA with CRT: Concrete results and practical countermeasures. Cryptographic Hardware and Embedded Systems-CHES 2002 pp. 81–95 (2003)
2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) CARDIS. Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer (2010)
3. Global Platform: Card Specification v2.2 (2006)
4. Hemme, L.: A differential fault attack against early rounds of (triple-) DES. Cryptographic Hardware and Embedded Systems-CHES 2004 pp. 170–217 (2004)
5. Hubbers, E., Poll, E.: Transactions and non-atomic API calls in Java Card: specification ambiguity and strange implementation behaviours. Dept. of Computer Science NIII-R0438, Radboud University Nijmegen (2004)
6. Iguchi-Cartigny, J., Lanet, J.: Developing a Trojan applet in a Smart Card. Journal in Computer Virology (2010)
7. Lanet, J.L., Bouffard, G., Machemie, J.B., Poichotte, J.Y., Wary, J.P.: Evaluation of the ability to transform sim application into hostile applications. Cardis (2011)
8. Oracle: Java Card Platform Specification
9. Piret, G., Quisquater, J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. Cryptographic Hardware and Embedded Systems-CHES 2003 pp. 77–88 (2003)
10. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: OPAL: An Open Platform Access Library. `http://secinfo.msi.unilim.fr/`
11. Smart Secure Devices (SSD) Team – XLIM, Université de Limoges: The CAP file manipulator. `http://secinfo.msi.unilim.fr/`

SmartCM A Smart Card Fault Injection Simulator

Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet, Julien Cartigny

SSD - XLIM Labs, University of Limoges,
83 rue d'Isle, 87000 Limoges, France,
{jean-baptiste.machemie, clement.mazin, jean-louis.lanet, julien.cartigny}@xlim.fr

***Abstract*—Smart card are often the target of software or hardware attacks. The most recent attack is based on fault injection which modifies the behavior of the application. We propose an evaluation of the effect of the propagation and the generation of hostile application inside the card. We designed several countermeasures and models of smart cards. Then we evaluate the ability of these countermeasures to detect the faults, and the latency of the detection. In a second step we evaluate the mutant with respect to security properties in order to focus only on the dangerous mutants.**

I. INTRODUCTION

Smart cards are devices prone to attacks in order to gain access to services or assets stored by the card. Several means have been used to retrieve these valuable information and recently fault injection appears to be the most efficient. Thus smart card manufacturers try to design countermeasures to embed in their operating system to prevent such attacks. Often solutions are based on dedicated code at the applicative level. We try here to evaluate the effect of a fault on smart card program in order to design efficient countermeasures. For that purpose we have developed our software fault injection simulator *SmartCM*.

Software fault injection is the process of evaluating software under anomalous circumstances involving external inputs or internal system state. It is often used to assess the correctness of a system design. Software fault injection tries to measure the degree of confidence that one can have in a given system by evaluating what could happen when faults are activated. Traditionally, the software-based fault injection involves the modification of the software execution on the system under analysis in order to provide the capability to modify the system state according to the programmer model view of the system. All sorts of faults may be injected, from the register, flags, and memory faults.

SmartCM modifies the EEPROM memory where the application is stored according to a fault model, examines the effect on the program and if the detection mechanisms embedded in the card are not able to discover the modification it saves the mutant application. Later, all the mutant applications are checked to decide if the mutation is dangerous or not.

The reminder of the paper is organized as follow. Section 2 presents an overview of the smart card attacks and defenses. Section 3 discusses *SmartCM*, an automated tool that we have developed to evaluate the fault propagation. Section 4 presents the experimental evaluation of *SmartCM* on industrial cases studies. Section 5 introduces our future developments and then we conclude.

II. BACKGROUND

A. Smart Card Attacks

Smart cards are objects commonly used in our daily life providing some computing capabilities and security features in a very small device. Examples of applications using smart cards are banking applications, electronic passport, health insurance card, pay TV, SIM card, etc. Therefore, they contain some sensitive information which must be protected against fraud. Since the beginnings, smart cards have suffered many hardware and software attacks in order to gain access to their assets.

Boneh, DeMillo and Lipton have proposed in [5] a new attack model against smart card which they called cryptanalysis in presence of hardware fault. This attack model initially focused on several public-key cryptographic algorithms like the RSA signature scheme and the Fiat-Shamir and Shnorr authentication schemes. It has been shown in [3] by Bihan and Shamir that DES is also vulnerable to these attacks. This has led to numerous forms of hardware attacks against smart cards using fault injection [14], [2].

Faults can be induced into the chip by the perturbation of its execution environment. Consequences of fault attacks can be perturbation of the chip registers (*e.g.*, the program counter, the stack pointer,...), or the writable memories (variables and code modifications). These perturbations can have various effects, and in particular, they can allow an attacker to gain illegally access to data or services if not detected. In the literature, we can find different manners to produce fault attacks but currently laser beam attack is the most difficult to tackle with.

B. Fault Model

To prevent a fault attack, we need to know its impact on the smart card. Fault models have already been discussed in details [4], [15]. We describe in the table I the different fault models in descending order in terms of attacker power. In this paper, we consider that an attacker can change one byte at a time *i.e.*: the precise byte error fault model. Sergei Skorobatov and Ross Anderson discussed in [12] an attack using the precise bit error model. But it is not realistic on current smart cards, because modern components implement hardware security on

Table I: Existing Fault Model

Fault Model	Precision	Location	Timing	Fault Type	Difficulty
Precise bit error	bit	total control	total control	bsr	++
Precise byte error	byte	total control	total control	bsr, random	+
Unknown byte error	byte	loose control	total control	bsr, random	-
Unknown error	variable	no control	no control	random	–

memories like error correction and detection code or memory encryption.

The process is the following, an attacker physically injects energy in a memory cell to change its state. Thus and up to the underlying technology, the memory physically takes the value `0x00` or `0xFF`. If memories are encrypted, the physical value becomes a random value (more precisely a value which depends on the data, the address and an encryption key). Then the attacker observes the effect of the fault characterized in terms of temporal and spatial parameters. If the result did not provide him any valuable information he restarts the process. This scenario validated with our industrial partner helps us to select the fault model: we choose the precise byte error. Thus, we assume that an attacker can:

- inject a fault at a precise clock cycle (he can target any operation he wants),
- only set or reset a byte to `0x00` or `0xFF` up to the underlying technology (bsr[1] fault type), or he can change this byte to a random value beyond his control (random fault type),
- target any memory cell he wishes (he can target a specific variable or register).

After defining the fault hypotheses we focus on the effect of the fault on a specific memory: the EEPROM. The system code is stored in ROM, it can suffer from a fault attack while the code is executed by the processor but this remains a transient fault which is more difficult to exploit for the attacker. At the opposite, the permanent error is the most valuable. This can occur when attacking the applicative code stored in EEPROM. But some recent smart cards have also system code stored in FLASH memory which may be subject to permanent error too. In both situations, modifying the code stored can change the behavior of the application leading to a potential aggressive application. Such a modified application, not detected by the embedded countermeasures is defined as a mutant.

C. Detection Mechanisms

There exists two different types of countermeasures against fault attack. The hardware countermeasures [1] which harden the ability to modify on-card programs and program flows. The software countermeasures in which software may check for faults or to ensure that no valuable information can be learned from injecting faults.

Hardware countermeasures include those which can be implemented by the industry to provide tamper resistant chips. In this category we can cite passive protections to increase the difficulty to succeed an attack (like random dummy cycles, bus and memory encryption, unstable internal frequency generators, etc.) and active protections that contain mechanisms checking whether tampering occurs and take countermeasures (generally the family of detectors like light detectors, supply voltage detectors, frequency detectors, etc). But those countermeasures are not dedicated to fault attacks and their detection ability is low. Currently software countermeasures are the most efficient solution.

Software countermeasures can be classified by their type.

- Cryptographic algorithm countermeasures which focus on the implementation of specific cryptographic algorithm and often provide better implementations of the cryptographic algorithms like RSA (which is the most frequently used public key algorithm in smart cards), DES, and hash functions (MD5, SHA-1, etc.).
- Applicative countermeasures; It is possible to implement at the application level several checks to ensure that the program always executes a valid sequence of code on valid data. It includes double condition checks, redundant execution, counter etc. and are well known by the developers. Unfortunately this kind of countermeasures increase drastically the program size. Because beside the functional code, it needs security code and the data structure for enforcing the security mechanism embedded in the application. Furthermore Java is an interpreted language therefore its execution is slower than with a native language, so this category of countermeasures suffers from bad execution time and add complexity for the developer.
- System countermeasures where protections are integrated directly at the system level. The main advantage is that the system and the protections are stored in the ROM memory, which is a less critical resource than the EEPROM and cannot be attacked. Thus, it is easier to deal with integration of the security data structures and code in the system. But often the design of an embedded Java Virtual Machine (JVM) relies on an offensive interpretation with a few system countermeasures and a robust Byte Code Verifier (BCV) that checks during load time the application.

Nevertheless all these countermeasures need to be evaluated in terms of efficiency, ability to detect a fault and cost: size of the memory footprint and execution time overhead.

D. Driving the Execution Mode on the JVM by the Application

In a previous paper [10], we have proposed a solution using a security feature available in Java Card 3.0 platform: the

[1] bit set or reset

annotations. But this approach is also fully applicable to Java Card 2.x platform using the custom component facility [13]. The idea is to drive the execution mode of the JVM by the application. The developer knows the semantics of its application and then can choose the best embedded countermeasure for each fragment of its code. Some of them are not sensitive while others need to be executed in the most secure mode. It is then possible to adjust the memories and CPU overhead to the optimum and reduce code in the application dedicated to countermeasure. The process is executed in two phases: outside the card we generate the annotations and in the card we change the execution mode according to the annotations. For example, the `@SensitiveType` annotation denotes that the whole method must be checked for integrity with the *check paths* mechanism.

```
@SensitiveType{
sensitivity= SensitiveValue.INTEGRITY,
proprietaryValue="PATHCHECK"
}
private void debit(APDU apdu) {
   if ( pin.isValidated() ) {
      // make the debit operation
   } else {
     ISOException.throwIt(
    SW_PIN_VERIFICATION_REQUIRED);
   }
}
```

For the off card part, we provide a tool that processes an annotated class file. It allows the use of several countermeasures like those defined in [11] (*i.e.*, Check paths, Basic Blocks, etc.) on methods or on byte code fragments. Then the file is securely loaded inside the card using the Global Platform protocol [8]. Java based smart cards can process custom components if it knows how to use them or else, silently ignores them. Of course to process the information contained in these custom components, the virtual machine must be adapted, for that purpose we have designed our own JVM. The virtual machine interprets the application code and while entering a method or class tagged with a security annotation, it switches to the required secure mode. This approach is compatible with non modified virtual machine.

With this mechanism, an attacker must simultaneously inject two faults at the same time on two different memory areas, one on the application code and the other on the system during the interpretation of the code. A dual fault is outside the scope of the chosen fault model and is not realistic according to the literature.

III. TOOL ARCHITECTURE

The simulation tool *SmartCM* aims at analyzing the effect of a fault on a Java Card program. Three different programs are used in this analysis. The first one is the mutation engine which takes as input a model of the card and the applicative program at the byte code level. It emulates the effect of the fault on the program according to a fault model and generates the mutant code. The mutant code is symbolically interpreted by the card and if an embedded countermeasure detects the deviant behavior the mutant is rejected, else it is stored as a mutant. The second tool is a risk analysis tool. If a mutant is generated we need to evaluate the impact of its behavior to decide if it is a hostile behavior or not. Then a last tool which is under development aims to integrate code into the application in order to reduce the number of mutants. We expect also to add (see future works) a module to recognize sensitive patterns during the development in order to provide a complete framework.

A. Smart Card Model

The smart card model integrates well known software countermeasures and specific ones developed in a previous thesis. It models also the nature of the memories used by the smart card. If the given smart card used an encrypted memory then the effect of a fault on the memory will be a random byte according to encryption algorithm. The second entry is a set of class files of the program. The user can choose a profile corresponding to registered smart cards.

B. Mutation Engine

The mutation engine is a brute force process which modifies the memory where the byte code is stored. The effect of the fault on the program is evaluated with an abstract interpreter that includes the management of the Java annotations. If the byte that has been impacted by the fault is an opcode, then according to the kind of memory (encrypted or not) the value of the new opcode is either `0x00` or `0xFF` or any value in this range. Then the mutation engine uses the smart card model to evaluate the execution of this new code for each values of the opcode and propagates the error until a countermeasure detects it (stack underflow, overflow, wrong local variable, wrong expected type,...). If a return of the method `ProcessAPDU()` (which can be considered as a main for a Java Card applet) of the applet is reached or an exception is never caught, then we consider that the mutant cannot be detected. We generate a class file that corresponds to the mutant code and the corresponding class file is stored for further analysis. Less secure is the card, more we must interpret the code and longer is the simulation. Parallelization is one of our current improvement in order to be able to analyze low-end smart card; each new code is independent from the others and then can be analyzed concurrently on different computers.

C. Risk Analyzer

The mutation of an application can generate several mutants according to the security of the platform. To help the programmer to understand the effect of the error, it outputs the original Java Code and the Java perspective (if possible) of the mutant code, it highlights the area where the code has been modified. Often the mutants are harmless but a security officer must check all of them. In order to facilitate this task we developed a risk analysis module that verifies a set of security properties on the mutant and decide to tag the mutants as

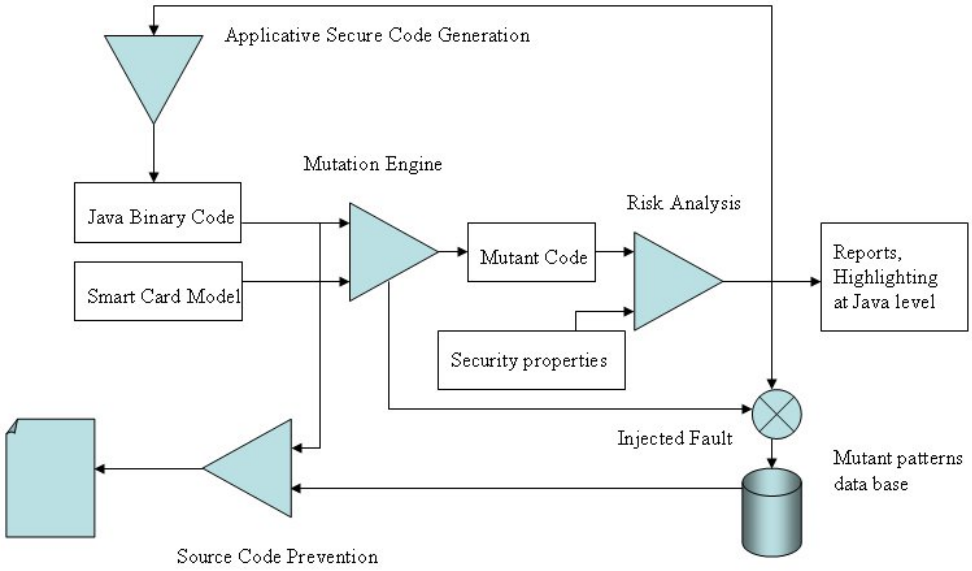


Figure 1: General architecture

dangerous or not. This tool can also include the possibility to add other security properties, to include an internal map of the the method addresses.

A mutant can be dangerous in several cases: it accesses objects and methods that are normally non authorized (potentially call to unwanted methods : `getKey()`), it changes its own internal behavior (*e.g.*: remove test), it performs action on its own data in an unexpected way.

1) Method calls: The issue is when another method is called or if one of the parameters has mutated. Methods must be invoked with the appropriate arguments (number and type), for that purpose we need to model the effect of the instruction on the operand stack and local variable array and verifies that it corresponds to the signature. If the number of arguments are not correct in terms of types on top of the stack a type confusion can occur inside the called method which is forbidden. For example, an address is given instead of a short and the short value is modified in the called method: it is the way to perform arithmetic on addresses which leads to a dangerous mutant. In terms of number of arguments: too few arguments is detected thanks to a stack underflow detection (if present in the card), too much arguments cannot be detected locally but will be detected by the caller. Normally at the end of a method the stack contains either zero element (return void) or one element that has the correct type.

The address of the called method is valid but different from the original. In fact, the linking process is done in the card. So the code to analyze do not have the real addresses (it contains only offset to the constant pool). According to industrial partners, it is possible to obtain the internal mapping of the methods (in the ROM or EEPROM area). In such a case, it is possible to generate the mutants where the address of a call is valid. We have to consider two cases, the arguments are valid (number and type) and it is a dangerous mutant or not and then it can be considered as harmless. If the mutation concerns an address that does not correspond to the structure of method, it will be detected by the JVM and thus it must not be considered as dangerous.

In case of `invokeInterface`, `invokeSpecial` and `invokeVirtual`, methods are invoked on another object. If the address does not correspond to a data structure of an object, the mutant will be detected during run-time, it is not dangerous. If it is a concrete address of an object that do not belong to the current security context, the mutant will be detected during run-time, thus it is not dangerous. If it is a concrete address of an object that belongs to the current security context, the method will be applied on the wrong object : the static analysis can not detect it, there is no possibility to have a dynamic map of the objects. We apply the same rule when accessing a field.

2) Changes in the control flow or data: The mutant can change the control flow of the method. The issue is that some methods are not called or the result is not evaluated. We must verify that all calls and all evaluations are performed because if one is skipped it is a dangerous mutant. The last point concerns the static fields or the local variables. If a local is used in an assignment *e.g.* `boolean testVariable = pin.isValid()`, and later this variable is used into a test, any assignment between definition and use can lead to a dangerous mutant.

IV. INDUSTRIAL CASE STUDY

A. Applications

Several Java Card applets have been used for the evaluation. Two SIM applets are representative of the type of code that a MNO (Mobile Network Operator) may want to add to their USIM Card. The first (*AgentLocalisation*) is oriented geolocalization services, this applet is able to detect when the handset (the device in which the USIM card is inserted) is entering or leaving a dedicated or a list of geographical dedicated cells (each cell is identified through a CellID value, which is stored on the USIM interface) and then sends a notification to a

dedicated service (registered and identified in the applet). The second (*OTP*) is more specialized to authentication services, the applet is able to provide a One Time Password (OTP) to the customer and/or an application in the handset. This OTP value is already shared and synchronized by the applet with a central server, which is able to check every collected OTP value using dedicated web services. We also evaluate a protocol payment applet designed by a major smart card manufacturer [7] and a hostile applet specifically designed to execute shell code when mutated [6].

B. Metrics

The collected metrics cover two different aspects: how efficient are the countermeasures and are they affordable. One can design very efficient code but if it does not fit the industrial requirements it remains useless. To fulfill the first one we need to verify their detection coverage and the latency of the detection. The second point is related with run time execution and memory footprint. It is widely accepted that an overhead over 5 % is considered as a maximum. So we need in a one hand to simulate the code (the objective of *SmartCM*) on the other hand to implement it in a JVM and evaluate its run time impact.

1) Evaluating Resources Consumption: The first category of metrics is the memory footprint and the CPU overhead. They have been obtained using the SimpleRTJ [9] Java virtual machine modified to accept multiple execution modes driven by annotations. This JVM targets highly restricted constraints devices like smart cards. The hardware platform for the evaluation is a board which has similar hardware as high end smart cards.

These metrics are very important for the industry because memories size directly impacts the production cost of a card. In fact, applications are stored in the EEPROM which is the most expensive component of the card. The CPU overhead is also important because most of the time, when challenging the card for some computation a quick answer is needed. So when designing a countermeasure for smart cards, it is important to have these properties in mind. To obtain the metrics in table II, all the countermeasures have been implemented on an embedded JVM that has similar properties as common smart cards.

2) Evaluating Mutants Detection: To evaluate the path check detection mechanism, we have developed an abstract Java Card virtual machine interpreter. This abstract interpreter is designed to follow a method call graph, and for each method of a given Java Card applet, it simulates a Java Card method's frame. A frame is a memory area allocated for the operand stack and the local variables of a given method.

The mutant generator has different smart card profiles:

- *The Basic Profile (BP)*: the interpreter executes, without running any checks, the instruction set. It corresponds to low end smart cards which rely entirely on the execution of BCV outside the card. Most smart cards corresponding to the 2.1 standard and some of the 2.2 standard fall into this category.
- *The Defensive Profile (DP)*: the interpreter checks that no overflow or no underflow occurs, that the used locals are inside the current table of locals, and that when a jump occurs it takes place inside the method. They consist in some verifications done by the BCV.
- *The Customized Profile*: this profile corresponds to the dynamically adjustable security of the JVM. The user can activate different countermeasures see [10] like the developed ones: path checking mechanism (PC), basic bloc integrity (BB), field of bits mechanism (FB) , or TCM mechanism. TCM is a detection mechanism that is not described in this paper and for which a patent is pending. For the purpose of this evaluation, we compare independently each countermeasure for the whole program. It is a pessimistic approximation because it is applied to the whole program, while it needs to be applied only on the sensitive methods or fragments of them.

3) Resources Consumption: Table II shows the metrics for resources consumption obtained by applying the detection mechanism to all the methods of our tested applications. The increase of the application size is variable, this is due to the number of paths that exist on a method. Even if the mechanism is close to 10 % overhead size and 8 % of CPU overhead, the developer can choose when to activate only for sensitive methods to preserve resources. This countermeasure needs small changes on the virtual machine interpreter if we refer to the 1 % increment. So, we can conclude that it is an affordable countermeasure.

Table II: Resources consumption for Customized Profile

Countermeasures	EEPROM	ROM	CPU
Field of bits (FB)	+ 3 %	+ 1 %	+ 3 %
Basic block (BB)	+ 5 %	+1 %	+ 5 %
Path check (PC)	+ 10 %	+1 %	+ 8 %
Typed CM (TCM)	0 %	<1 %	+<1 %

4) Mutant Detection and Latency: The obtained results show the efficiency of the developed countermeasures. The table III shows the generated mutants in each mode of the mutant generator for five applications. The table IV shows the latency which can be defined as the number of instructions executed between the attack and the detection. With the basic profile, no latency is recorded because no detection is made. This value is really important because if a latency is too high maybe instructions that modify persistent memory like: `putfield`, `putstatic` or an invoke instruction (`invokestatic`, `invokevirtual`, `invokespecial`, `invokeinterface`) can be executed. If a persistent object is modified then it is manipulated during all future sessions between the smart card and a server. So this value has to be as small as possible to reduce the chances of having instructions that can modify persistent memory or send data to the reader.

Path check (PC) fails to detect mutants whenever the fault that generates the mutant does not influence the control flow of the code. Otherwise, when a fault occurs that alters the control flow of the application then this countermeasure detects

Table III: Results - Mutation

	BP	DP	CP			
			TCM	FB	BB	PC
Wallet (470 inst.)	440	54	30	10	0	37
Otp (4568 inst.)	7960	464	378	40	0	1032
AgentLoc. (3504 inst.)	6486	356	343	10	0	784
Payment (1100 inst.)	2140	304	250	1266	0	2140
Hostile (825 inst.)	1622	0	50	14	0	76

Table IV: Results - Latency

	BP	DP	CP			
			TCM	FB	BB	PC
Wallet	-	2,91	2,92	2,43	2,72	2,42
Otp	-	3,64	3,56	8,61	12	17,18
AgentLoc.	-	11,8	12,1	2,43	10,20	13,06
Payment	-	5,54	6,55	0,5	-	-
Hostile	-	-	0,77	0,71	-	0,79

it. With this countermeasure it becomes impossible to bypass systems calls like cryptographic keys verification. Basic bloc is the most efficient countermeasure.

5) Risk Analysis: The risk analysis tool is able to cut for the configuration the AgentLocalisation applet using the TCM, the 343 mutants to 8 dangerous mutants which reduces drastically the effort for manual inspection of the code. It needs only a few seconds to execute this analysis, in the case where the internal mapping is provided. The tool is able to classify between 90% to 97% of the mutants into the non dangerous category.

V. FUTURE WORKS

In the next steps, we will try to improve the development phase. We want to analyze the mutant codes, to deduce at the Java level which structures are sensible for a given Java Card platform. Then we will propose to generate the code that could (or not) eliminate the mutant. For example, if the mutant is generated thanks to a modification of the control flow, then a double conditional can solve this issue or a sequence counter. In some cases it is possible to generate automatically applicative counter measures. The objective is to pinpoint the sensitive structures, suggest a countermeasure or a warning. The second improvement concern the development phase, close to the previous one but using learning networks. While coding its application the developer could we warned that a given pattern (at the byte code level) is sensitive. Each time an application is analyzed, we enrich the data base of sensisitive patterns. Each sensitive pattern is associated to Java source code, if possible.

VI. CONCLUSIONS

In this paper, we presented a simulation approach to evaluate the impact of a fault on the program memory of a Java based smart card. We define a platform model through a profile then we generate all the possibilities of an error. The collected metrics during the simulation are used to adopt a given counter-measure: ability to detect a fault, latency of the detection, cost in term of memory (ROM, RAM and EEPROM), cost in term of CPU overhead. We have developed our own countermeasures and evaluate them in this context. The designed tool offers also the possibility to eradicate non relevant mutants and view only those that endanger the security of the platform. it provides the ability to read it either at the byte code level for experts but also (if possible) at the Java language level.

Designing counter measures takes into account the compliance with previous file formats according to either the Java Card 2.x or 3.x specifications. Our countermeasures are affordable due to a low memory footprint. For the applet designer, he can adjust dynamically the security level of the JVM according to the semantics of its program. The developer knows exactly which program fragment needs to be protected and which can be executed without specific treatment.

With this framework, both the developer and security evaluator can take decisions concerning the security of the smart card application. For the developer company, reducing the size of the embedded code minimizes the cost of the application. For the security evaluator it provides a semi-automatic tool to perform vulnerability analysis.

REFERENCES

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, and I. Rehovot. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[2] G. Barbu, H. Thiebeauld, and V. Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. *Smart Card Research and Advanced Application, Cardis 2010*, LNCS 6035:148–163, April 2010.

[3] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. *Lecture Notes in Computer Science*, 1294:513–525, 1997.

[4] J. Blomer, M. Otto, and J.P. Seifert. A new CRT-RSA algorithm secure against Bellcore attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320. ACM New York, NY, USA, 2003.

[5] D. Boneh, R.A. DeMillo, and R.J. Lipton. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[6] G. Bouffard, J-L. Lanet, and J. Cartigny. Combined software and hardware attacks on the java card control flow. In *Proceedings of Cardis Tenth Smart Card Research and Advanced Application Conference*, 2011.

[7] P. Girard, J-L. Lanet, A. Plateaux, and K. Villegas. A new payment protocol over the internet. In *CRISIS, International Conference on Risks and Security of Internet and Systems*, pages 51–56, 2010.

[8] Global Platform. Official web site, http://www.globalplatform.org, 2010.

[9] Simple RTJ. Official web site, http://www.rtjcom.com, 2010.

[10] A. Sere, J. Cartigny, and J-L. Lanet. Automatic detection of fault attack and countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security*, pages 1–7. ACM, 2009.

[11] J. Sere, A. Cartigny and J-L. Lanet. A path check detection mechanism for embedded systems. *Proceedings of SecTech 2010*, 6485:459–469, 2010.

[12] S.P. Skorobogatov and R.J. Anderson. Optical fault induction attacks. *Lecture notes in computer science*, pages 2–12, 2003.

[13] SunMicrosystems. *Java Card 3.0.1 Specification*. Sun Microsystems, 2009.

[14] E. Vetillard and A. Ferrari. Combined attacks and countermeasures. *Smart Card Research and Advanced Application, Cardis 2010*, LNCS 6035:133–147, April 2010.

[15] D. Wagner. Cryptanalysis of a provably secure crt-rsa algorithm. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 92–97. ACM New York, NY, USA, 2004.

Hot Updates for Java Based Smart Cards

Agnes C. Noubissi, Julien Iguchi-Cartigny, Jean-Louis Lanet
Department of Computer Sciences, University of Limoges
83 rue d'Isle, Limoges, France
agnes.noubissi@xlim.fr
julien.cartigny@xlim.fr
jean-louis.lanet@xlim.fr

***Abstract*—Systems need to be updated in order to correct vulnerabilities, fix bugs but also to enhance functionalities. Traditional software update mechanisms usually stop the software that need to be updated, apply the update then restart the system. However, this approach is not appropriate in critical systems such as banking or telecommunications.**

A solution is the dynamic software update (DSU) approach which patches the software on-the-fly (*i.e.* during execution). Several mechanisms have been proposed to offer this functionality to native and virtual machine environments for desktop and server computers. But using DSU in embedded systems (and especially in smart cards) raises several challenges regarding the resource and security constraints.

In this paper, we present our work, EmbedDSU, a framework for DSU on Java based smart cards. The first part of EmbedDSU is an off-card mechanism which computes the changes between two versions of classes. This limits the update process only on the part of the component which is affected by the modification. The second part of EmbedDSU is the interpretation od the changes on-card and update of the methods in the class definition and object instances.

I. Introduction

To cope with new, modified or extended functionalities, software must be updated, in order to remove bugs or improve functionalities. A simple update approach is to stop the old version of the software (which could require the whole system to shut down), to apply the update then to restart the new version. However, this approach is not adequate for applications which must run continuously even during the update process.

Dynamic Software Update (also called DSU, on-line update, on-the-fly or hot update) is the process of updating an application or a software component without stopping neither the system on which it is executed, nor the application itself. There is already existing literatures about DSU on hardware or software approaches.

In this paper, we present EmbedDSU, a software-based DSU technique for Java based smart cards which relies on the Java virtual machine. Our approach is based on the modification of an embedded virtual machine to offer dynamic update functionalities. EmbedDSU is divided in two parts : off-card and on-card. In off-card, in order to apply the update only on the parts of the application or component that are really affected by the modification or by the update, we have implemented a DIFF generator which determines and expresses the syntactic changes between versions of classes. Then, the DIFF file result is transfered to the card and used to perform the update.

Under some specific assumptions, the approach permits generic updates of Java applications or Java components and has the following properties:

1) No human intervention required: The process is fully automatic: given two versions of a class, the DIFF generator is able to compute and express the changes between the two versions for class interfaces, fields, method bodies and constant pools or class meta-data.

2) Support any granularity of change: This mechanism also supports various update granularity: fields, method signature, one or a set of instruction blocks, and related classes.

3) Atomicity of update: To ensure coherence of the system, update is performed atomically.

4) Semantic preservation: After update, the updated classes obtained by using the DIFF have the same behavior as if it were transfered entirely and linked into the card.

To validate our approach, we implemented and tested EmbedDSU with an existing embedded Java virtual machine called SimpleRTJ (Simple Real Time Java)[17]. We have modified SimpleRTJ and adapted it, in order to offer DSU mechanisms. In this paper, we present our motivation, explain the update process architecture and describe our implementation. Then we present the first results obtained using our implementation and potential future works.

II. Motivation and Background

A smart card is a piece of plastic with the size of a credit card, in which a single chip microcontroller is embedded. Usually, microcontrollers for cards contain a microprocessor and several kind of memory: RAM (for run-time data and OS stacks), ROM (in which the operating system and the romized applications are stored), and EEPROM (in which the persistent data are stored). Due to the chip size constraints, the amount of memory is small. Most smart cards sold today have at most 5 Kbytes of RAM, 256 KB of ROM, and 256 KB of EEPROM. The communication uses a half duplex serial line with a very low level protocol: APDU. The card acts as a server and expects a request from a reader.

A smart card can be viewed as a secure data container, since it securely stores data and it is used securely during short transactions. Its safety relies on the underlying hardware: this chip usually also contains some sensors (like light sensors, heat sensors, voltage sensors, etc.), which are used to disable the card when it is physically attacked.

Modern smart card embeds a virtual machine which interprets codes already romized with the operating system or downloaded after issuance. The standard Java Card 3.0 [5,6] is an exemple of virtual machine specification for smart card, available in two editions: Classic Edition or Connected Edition. A dedicated Java virtual machine is called Java Card virtual machine (JCVM). It interprets Java Card applications and manages access to smart card resources, thus serving as the smart card operating system. The JCVM has he ability to run multiple applications, sometime uploaded in the card after it has been issued. But the JCVM is a not-stop VM: the life cycle is the card life because persistent objects are preserved even after expiration of the communication sessions with the reader. Thus the JCVM runs continuously and then update must be applied at runtime.

The security of the plateform from the sofware point of view relies on several mechanisms. First, the ability to download code into the card is controlled by a protocol defined by Global Platform [16]. This protocol ensures that the owner of the code has the necessary credentials to perform such an action. Because, Java Card is an open platform for smart cards, i.e. able of loading and executing new applications after issuance, different applications from different providers run in the same smart card. Thanks to type verification, the byte codes delivered by the Java compiler are safe, i.e. the application loaded is not hostile to other applications. Furthermore, the Java Card firewall checks permissions between applications in the card, enforcing isolation between applications.

Java Card is a subset of Java, with no multithreading, simple type (no float) only one-dimensional array, no garbage collection and so on. Applications are relatively simple to develop and are based on the Java Card Applet model. The main method process is invoked while receiving incoming bytes and according to the content of the header executes the request and sends back data.

Currently, it is possible to provide update of applications on Java based smart cards by replacing old version by new one, an operation using the post-issuance mechanism. But for this, the application needs to be stopped before the update, which is impossible for Java system components because they are romized. Our goal is to patch the API component which are store un ROM. With indirection table, we can replace part of the API using the EEPROM area.

In the case of E-passport, if weakness in a cryptographic algorithm is discovered, the actual solution is to return all of them to the issuer. This solution is too complex when there is an important number of E-passports on the field. This is not an hypothetical threat. Indeed, recent exploits have been shown on the MIFARE chip [3] (a contact-less smart chip technology based on ISO 14443). For instance Pltz and Karsten [11, 12] described a partial reverse-engineering method for the MIFARE classic chip. Similarly, Gans and *al.* [13] presented a technique to manipulate the contents of a MIFARE classic card, and then compute some cryptographic keys, PIN code, or modify some security tests.

So, it is necessary to provide a mechanism to dynamically patch the system components when the chip is inside a terminal or near a Near Field Communication (NFC) reader. Java card 3.0 is the main target for EmbedDSU.

III. The proposed architecture

The architecture (Fig. 1) of our system is divided into two parts: on-card and off-card.

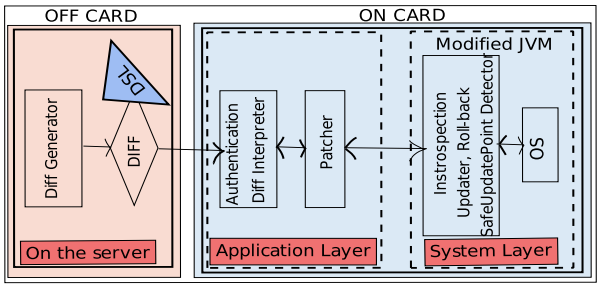


Fig. 1. Workflow of the dynamic update using EmbedDSU

A. Off-card

The first step is to compute the changes between two versions (the original and new one) of the system component module to update, with the goal to found the parts of the component that are modified. EmbedDSU uses a DIFF generator to generate a DIFF file. This file contains the changes between two versions of classes, expressed using a Domain Specific Language (DSL). Then the DIFF file is parsed and converted into a compressed binary format to minimize the size and to reduce the interpretation cost by the card. Finally, the binary format file is transferred to the smart card for patching.

B. On-card

1) Application Layer: The binary DIFF file is uploaded into the card. After signature check with the wrapper, the binary DIFF is interpreted and the resulting instructions are transfered to the *Patcher* in order to perform the update. The *Patcher* has the role to initialize update data structures. These data structures are read by the updater module to know what to update and how to update, by the *safeUpdatePoint* detector module to know when apply the update and by the roll-backer to know how to return in the previous version. All these issues pass through introspection and modification of data structures of the virtual machine, this requiring Virtual Machine modification.

2) System Layer: The last element of EmbedDSU is the modified virtual machine to support the followings features: (1) Introspection module which provides search functions to go through VM data structures like the references tables, the threads table, the class table, the static object table, the heap and stack frames for retrieving necessary informations to other modules;

(2) *updater* module which can modify and update object instances, method bodies, class metadata, references, affected registers in the stack thread and affected VM data structures;

(3) *SafeUpdatePoint* detector module which permits to detect safe point in which we can apply the update by preserving coherence of the system;

(4) *Rollback* module which permits to obtain previous versions of codes, instances, stack thread and VM data structures.

All these modules interact to provide DSU functionalities.

IV. Implementation

A. Off-card: Design of Diff generator

It exists several tools able to compute textual differences between files. However theses tools do not consider the changes in program behavior which can be caused indirectly such modifications We defined a differencing algorithm called DIFF generator that uses the control flow graph (CFG) in order to identify changes between two versions of program and then compute syntactic patches.

Indeed, DIFF generator can handle object-oriented features, capture the behaviors of object-oriented components, and identifie changes between versions. It takes as input an original class file C and modified version C' of the class file to update and generate a DIFF file formatted with a DSL [18] defined for the purpose.

It starts by generating the corresponding arborescence structure of the two classes (*.class*) using BCEL [15] and then performs the comparison first at the class interface and field levels, then at the method level. It matches method of the class file C with methods having the same name in C' to obtain a set of method pairs. For each pair, it compares signature, interfaces, local variables, exception tables, and finally, bytecode instructions. And, for bytecode instruction, it constructs the corresponding control flow graphs in order to model behaviors.

Each CFG represents the control flow of the corresponding method, with node representing basic blocks and egdes representing possible flows from one basic block to another. Basic block, in this case, is a set of bytecode related to a statement (or instruction) which can be entered only at the beginning and leaved at the end. For comparing the two CFGs G and G' of methods m and m', the goal of our algorithm is to find, for each statement in m, a matching or corresponding statement in m', based on the method structure. Thus, it requires to find for a basic block of G, the corresponding one in G' and then compare their contents, so compare relative statements to identify differences and similarities between them. Hence, the DIFF generator can provides the list of added, deleted or modified instructions with appropriate information about their location and attributes (for example their parameters which can be an *index* in the constant pool). In general, at the methods and fields level, the rule is: **methods or fields in C that do not appear in C' are deleted, whereas those in C' that do not appear in C are added**. Furthermore, the DIFF generator also compares the entries of constant pools of two classes to obtain information in order to modify class metadata of the old version in order to correspond to the new version.

In conclusion, DIFF generator can detect added, deleted or modified constant pool entries, class interfaces, fields, method signatures or interfaces, local variables, bytecodes or method bodies and exceptions. Thus, produces Diff file to transfer in on-card for the update.

B. On-card

In on-card, the main element is the modified virtual machine which contains DSU modules. This modified virtual machine can have three modes of execution.

- The mode before detecting a new version of an application, called the *standard mode*,
- The mode after receiving the DIFF file and before detecting a safe point or a quiescent state, called the *QuiescentStateSearch mode*,
- The mode after detecting the safe point, called the *update mode*.

1) Standard mode: During this mode, the virtual machine works normally without DSU process overhead. However, we need to know how the notification about the new update can be provided to the process managing updates (update manager). In our approach, the knowledge of an available new update is obtained when the smart card is powered up by contact or contacless reader. So, if an update is available, the reader can send information to the update manager which can load the corresponding DIFF file and stored it in persistent memory in order to be used later to perform update. To search that instant, the virtual machine switches to *QuiescentStateSearch* mode. We will discuss about centralized repository in the section future works.

2) QuiescentStateSearch mode: To obtain a safe point, we need to introspect the corresponding component state. In our approach, it can be search at any point in time during execution. It consists of:

- Classes. It is set of classes related to class to be updated (their method bodies and class metadata).
- Frames. It means all frames in the execution stack related to a method of the class to be updated. Knowing that, for each call of a method, a frame is created on the execution stack and it contains references to local variables, method parameters and a program counter which reflect the lines of method's bytecode.
- Objects. These are the active instances of the class to be updated present in the heap of the virtual machine. Active objects mean objects that are referred from method frames on execution stack or referred by a field of any other active instances.

Indeed, detecting safe update point is very important in the case of online update because of the likelihood of a system crash that will leave the system in an unstable state. For example, if an update of object instance occurs while we continue to access to the deleted attributes non longer present in new object instance in the heap. And then, we defined the safe point state, like a state in which we do not have frames related to a modified method in the stack thread.

So, when switching to the *QuiescentStateSearch*, all calls to a class method to be modified are blocked, in order not to

have others frames related to a method to be modified in the stack thread. Nevertheless, others frames can continue to be created on the stack. To determine a safe point, only frames of the component state are used. At starting point, we count all frames related to a modified method present in the stack thread. If the value is not equals to zero, then the update is delayed, the virtual machine can continue to execute others applications. However, the value is decremented each time, these methods finished their execution. When the value equals zero, then the safe update point is obtained and the virtual machine can switch to the update mode.

Moreover, it is necessary to take into account the case where the safe point is never reached. And for that, in parallel of searching safe point, we initialized an int value which represent the time counter. This time counter permits to wait for a fixed waiting time. If the fixed time is reached, then we stop the search of safe point and switch to the standard mode. The card manager then sends a message to the reader indicating the impossibility to update the card.

3) Update mode: In this state, update must be done in an atomic way, so updates are either completely or not applied at all. The goal is to modify the component state of the old version to obtain the new one corresponding to the new version.

Update process starts by updating method bodies and class metadata of the class to be updated and related classes. Indeed, with the DIFF file of the class to be updated, we can know entries of constant pool to modify and for each modified method, parameters, local variables and bytecode to modify. Then, for updating the code of the class, updater module copies while modifying the old version of class metadata like constant pool, field table, method table, constant table and for each method that does not deleted, it copies while modifying method header, and bytecode instructions. So, the old version is transfered to a new space while modifying to obtain the new one.

After updating the class, update process continues with the update of all other constant pools of related classes to modify references to old methods or fields in order to point to the new field entry table, or to point to the new methods.

When updating a class C to C', we must ensure that all instances of class C are updated to be instances of class C'. For updating instances, garbage collection approach is used. It consists of a heap traversal from persistence root - which include object references in reference table, statics and stacks local variables- and for each encountered object O which belongs to C or which contains a reference to class C, need to be updated. The modified garbage collector contained by the Updater module, creates a new object O' with the appropriate size, rewrites unmodified fields O'.f with value of O.f and initializes added fields of O' with initial values computed off card. Afterwards, the Updater saves the new object references in the reference table of the VM. The old version of an instance is deprecated immediately after the new version has been created and filled with values, but the old object references is also saved in order to be used for roll-back if necessary.

Update process continue by updating references in frames to point to the new object instances and return addresses to point to the method bodies in the new space of the class. After realising the update, garbage collector is called to free the memory space.

4) Roll-Back: When there are errors at any point during update, in order not to leave any incoherence in the application, the update process must stop and the application must continue with previous component execution state. Indeed, roll-back functions allow to revert back to:

- The old version of instances by using old object references saved during instance updates and replace new references by old.
- The old version of code by changing all references in class metadata of related class to point to the old methods and fields in the old space of the class.
- The old version of object references and return addresses in the stack frames. And of course others VM data structures.

V. Preliminary Benchmarks

To determine EmbedDSU instance updates time, we compare cost of the updated garbage collector to the normal garbage collector. So, we determine the perfomance during instance updates when we added, deleted or re-ordered fields in the class to be updated and we compare to the normal garbage collector perfomance. The table 1 describes the different instance versions or configurations used for the micro benchmarks.

TABLE I
THE THREE INSTANCE VERSIONS USED

Re-ordered fields	Added fields	Deleted fields

class C' {
int var2;
short var3;
int var1;
Object var4;
}
	class C' {
	short var5;
	int var1;
	int var6;
	int var2;
	short var3;
	Object var4; }
		class C' {
		int var1;
		short var3;
		}

The benchmark have three classes with re-ordered fields, added fields or deleted fields. The original class contains four fields (two integers, one short and one reference to an object initialize to null). We measure the cost of perfoming update while varying the number of objects for each configuration. We create 400 objects of the class to be updated and for each fraction of objects between 0% and 100%. So, after each set of 40 objects, we execute each configuration 25 times and report the median time obtained. And we realise the same for normal garbage collector but without dead objects, so just run through the heap of the VM from persistance roots.
Figure 2 shows the results.

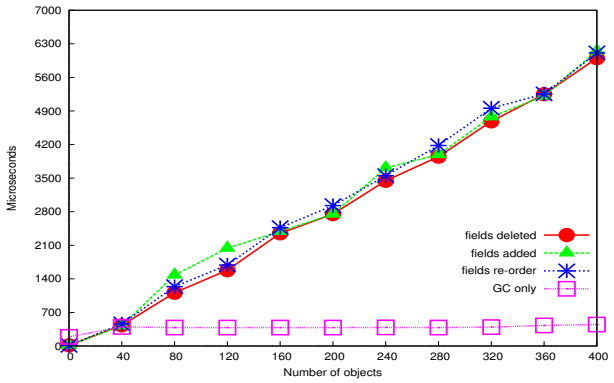


Fig. 2. Benchmarks results for changing the fields of a class compared to a garbage collection run

We need to precise that normal garbage collector is applied without dead objects in the heap and then, without overhead brings by space memory free. In this case, the normal GC is highly optimized, whereas update instances, have overhead due to others functions like transformer functions which copie one field at a time and initialize new fields with values provided in off card and specified in the DIFF. This extra functions make the process more expensive compared to this normal GC.

In other hands, we can see that adding fields is the most expensive operation under all others instances modifications because, it needs to search a space memory with the new appropriate size and then copies the old mapped fields and initializes the new ones by the initial values provided. Re-ordering instance fields is also costly because we need to create a new object, make a copy of the fields one-by-one for all instances, but in this case, we do not need to read initial values provided by the DIFF and this can reduce cost compared to adding fields operation. However, deleting instances field remains the least expensive operation, the objects are copied to their new space skipping the deleted fields and we do not need to initialize some fields with the DIFF.

VI. Related works

Several dynamic Java software update techniques have been presented in the literature [2, 7, 8, 9, 10]. Al. Orso and An. Rao [10] have presented a technique based on class renaming and code rewriting, which then performs the upgrade by dynamically swapping the classes at runtime through a tool called DUSC. However, the class renaming is a time consumming process and there is a space overhead in the case of changes in the class hierarchy (renaming, inserting or deleting classes).

Suriya Su., Michael H. and Kathryn S. have presented JVOLVE [7], a Java virtual machine which supports online softwares update. They use an Update Preparation Tool (UPT), the Jikes RVM dynamic class loader, a JIT compiler and class transformer functions to perform dynamic update.

Our approach follows this idea, but instead of use UPT to get the list of direct or indirect modified classes, we use a tool to express the real changes in the content of classes, so obtain the real modifications between two versions of classes for example and represent them using a DSL designed specifically for capturing differences between classes in order to save efficiently these changes.

JDrums [8] and DVM (Dynamic Virtual Machine) [9] have modified their virtual machine to support dynamic update like JVOLVE. However, JDrums performed a lazy update from DVM and JVOLVE. Indeed, JDrums use to check objects pointer dereference to determine if a new objects class is available and this technique brings about an additional overhead.

Nowadays it is not possible to dynamically update system components on the card. But, we can use one of the techniques described for non embedded systems, however we can note that these existing techniques have been tested in platforms, which are different of the smart card platform in the regard of resource constraints.

VII. Future works

A. Full State transfer

Currently in EmbedDSU, for updating instances, the strategy for initializing fields consists of searching fields that match by their name and type. And for matching fields, we copy the values from the old to the new instance. All others fields are initialized with a constant value obtained in off-card and then provided by the DIFF. But, sometimes, initial values can be a result of a method execution or a result of an expression to evaluate during execution. Then, it is necessary to provide dynamic initialization of new instances fields during update process in order to apply an instance full state transfer by providing state transfer functions.

For implementing that feature, we expect to couple state transfer file with the DIFF file in on-card, with state transfer functions provided by the developers. We believe that developers themselves have the best understanding of evolutionary changes in their applications. However, to not allow human intervention in the process, the automatic way will be better.

B. Late loading of DIFF file

In this section, we discuss how the notification of available new updates can be provided to on-card update manager. We propose to use a centralized repository, where all updates (DIFF files) are stored. The on-card update manager can then, for an application or a system component, check whether any new versions are available. However, this solution requires that in the repository, we have a service registry that is able to give for a precise smart card, all installed applications and system components. Currently, the question is how to implement the update manager in order to check and load updates or DIFF files in order to apply dynamic update? For simplicity, we assume that, all card readers can have some primitives update informations about DIFF file which are really stored on the centralized repository. Then, when the reader powered up the card, it can send update message to on-card update manager to inform it that new updates are available. After that, on-card

update manager can provide a push request for obtaining a set of DIFF files related to classes to be updated.

C. Dynamic wrapper methods or proxies methods

Currently, EmbedDSU permits update in cascade of a class and related classes. But, this approach can made the card unavailable during a certain time for the users if we have an important numbers of class to update. Therefore, we think to update only related classes which contains calls to deleted methods but for other classes which contains only modified methods including the method signature, we expect to provide dynamic wrapper or proxies methods created during update. Those wrappers forwarded old method calls for corresponding to the new calls with real number and type of parameters. However, we need to precise that all those signature modifications can not always be wrapped. We can have many type of changes, for instance a method RT m(T1 p1, T2 p2) can be modified to perform change in:

- Orders of parameter : $\rightarrow$ RT m(T2 p2, T1 p1), wrapper method can be such as :
  RT m(T1 p1, T2 p2) { **return** m(p2, p1); }
- Less of parameters : $\rightarrow$ RT m(T2 p2), in such case, wrapper can look as :
  RT m(T1 p1, T2 p2) { **return** m(p2); }
- Additional parameters : $\rightarrow$ RT m(T1 p1, T2 p2, T3 p3).

That later case is the very difficult, because we can not generate always a initial value for the added parameters. Indeed, for parameter like password, secret code, or pin code, if that type of parameter is added, we can not predict the value to affect. But for some cases, in which initialization is possible, we can then have a wrapper similar to :

RT m(T1 p1, T2 p2) {
T3 p3 = null ; // or other initial value depending on type of parameter;
return m(p1, p2, p3); }

For static or final methods where behavior cannot be modified, it is little more complicated. Our assumption is to require that new versions always follows the inheritance hierarchy of the old version.

VIII. Conclusion

In this paper, we present our on-going work EmbedDSU, a technique for dynamic update code in the context of Java based smart card. We show that EmbedDSU is divided in off-card and on-card mechanism and allow arbitrary modifications to Java classes in on-card through the DIFF file obtained in off-card. We need to precise that our benchmarks are done on an AMD Athlon Dual core machine, running at 1 GHZ with 2GB of RAM on which Ubuntu kernel version 2.6.28. But, we apply some constraints like using 2MB for heap of the VM. However, currently, we transfer the modified virtual machine to an evaluating board AT91 EB40A [14] to have the realistic benchmarks related to execution time for transfering the DIFF, interpret it, searching the safe point, applying instance updates to compare to the time to transfer a new version of a class, link and continue execution. And determine also memory used during those operations.

References

[1] Agnes C. Noubissi, Jean-Louis Lanet and Julien Iguchi-Cartigny, *Incremental Dynamic Update For Java Smart Cards Applications*, ICONS'10, France, April 2010.

[2] Jonathan T. Moore, Michael Hicks, and Scott Nettles, *Dynamic software Updating*, Programming Language Design and Implementation (PLDI), ACM, 2001.

[3] Semiconductors Austria GmbH Styria, *http://www.mifare.net/*

[4] Zhiqun Chen, *Java Card Technology for Smart Cards*, Addison, Wesley, 2000.

[5] *The Java Card 3.0 specification*: http://java.sun.com/javacard/

[6] Milan Fort, *Smart card application development using Java Card Technology*, SeWeS 2006.

[7] Suriya Subramanian, Michael Hicks, Kathryn S. McKinley, *Dynamic Software Updates : A VM-Centric Approach*, PLDI, June 2009.

[8] Jesper Andersson and Tobias Ritzau, *Dynamic deployment of Java applications*, Java for Embedded Systems Workshop, London, May 2000.

[9] Earl Barr, J. Fritz Barnes, Jeff Gragg, Raju Pandey and Scott Malabarba, *Runtime support for type-safe dynamic java classes*, ECOOP, 2000.

[10] Alessandro Orso, Anup Rao, and Mary Jean Harrold, *A technique for dynamic updating of Java Software*, ICSM, 2002.

[11] Karsten Nohl, David Evans, Starbug and Henryk Pltz, *Reverse-Engineering a Cryptographic RFID Tag*, USENIX Security Symposium, San Jose, CA. July 2008.

[12] Karsten Nohl and Henryk Pltz, *MIFARE, Little Security, Despite Obscurity*. Presentation on the 24th Congress of the Chaos Computer Club in Berlin, December 2007.

[13] G. de Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia, *A Pratical Attack on MIFARE Classic*, CARDIS, 2008.

[14] *ATMEL Corporation* : http://www.atmel.com/products/AT91/

[15] *Apache* : http://jakarta.apache.org/bcel/

[16] *Global Platform* : http://www.globalplatform.org/

[17] *SimpleRTJ* : http://www.rtjcom.com/

[18] Arie van Deursen, Paul Klint and Joost Visser, *Domain-specific languages: an annotated bibliography*, June 2000.

International Journal of Security and Its Applications
Vol. 5 No. 2, April, 2011

Evaluation of Countermeasures Against Fault Attacks on Smart Cards

Ahmadou A. SERE
SSD - XLIM Labs,
University of Limoges
JIDE, 83 rue Isle,
Limoges, France
ahmadou-al-khary.sere@xlim.fr

Julien Iguchi-Cartigny
SSD - XLIM Labs
University of Limoges
JIDE, 83 rue Isle,
Limoges, France
Julien.cartigny@unilim.fr

Jean-Louis Lanet
SSD - XLIM Labs,
University of Limoges
JIDE, 83 rue Isle,
Limoges, France
Jean-louis.lanet@unilim.fr

Abstract

Java Card are devices subject to either hardware and software attacks. Thus several countermeasures need to be embedded to avoid the effects of these attacks. Recently, the idea to combine logical attacks with a physical attack to bypass bytecode verification has emerged. For instance, correct and legitimate Java Card applications can be dynamically modified on-card using laser beam. Such applications become mutant applications, with a different behavior. This internal change could lead to bypass control and protection and thus should offer illegal access to secret data and operation inside the chip. In this paper, we propose a set of countermeasures that can be activated by the developer using the annotation mechanism. These countermeasures are efficient but also affordable for the smart card domain, as shown by the evaluation of the coverage and memory usage.

Keywords: *Smart Card, Java Card, Fault Attack, Control Flow Graph*

1. Introduction

A smart card can be viewed as a secure data container, since it securely stores data and it is securely used during short transactions. Its safety relies first on the underlying hardware. To resist probing an internal bus, all components (memory, CPU, crypto processor...) are on the same chip, which is embedded with sensors covered by a resin. Such sensors (light sensors, heat sensors, voltage sensors, etc.) are used to disable the card when it is physically attacked. The second security barrier is the software. The embedded programs are usually designed neither for returning nor modifying sensitive information without guaranty that the operation is authorized.

Java Card is a kind of smart card that implements the standard Java Card 3.0 [10] in one of the two editions "Classic Edition" or "Connected Edition". Such smart cards embed a virtual machine (called Java Card Virtual Machine or JCVM), which interpret application bytecodes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [7]. This protocol ensures that the owner of the code has the necessary credentials to perform the action.

Java Cards have shown improved robustness compared to native applications regarding many attacks. They are designed to resist to numerous attacks using both physical and logical techniques. In the category of hardware attacks, the most powerful attacks are hardware

based and particularly fault attacks. A fault attack modifies part of memory content or signal on internal bus and lead to deviant behavior exploitable by an attacker (a comprehensive consequence of such attacks can be found in [9]). Although fault attacks have been mainly used in the literature from a cryptanalytic point of view [2, 8, 11], they can also be applied to the executable code embedded in the device. For instance, while choosing the exact byte of a program the attacker can bypass countermeasures or logical tests. We called *mutants* such modified application.

Designing efficient countermeasures against fault attacks is important for smart card manufacturers but also for application developers. For the manufacturers, countermeasures must have the lowest cost in term of memory and processor usage. These metrics can be obtained with an evaluation on a target [12]. For the application developers, they have to understand the ability of their applets to become mutants and potentially hostiles in case of fault attacks. Thus, the coverage (the number of mutants which can be potentially generated by fault attacks) and the detection latency (the number of instructions executed between an attack and its detection) are the most important metrics. In this paper we present a workbench to evaluate the ability of a given application to become a hostile applet with respect to the fault hypothesis and the different implemented countermeasures.

The rest of this paper is organized as follows: first, we introduce a brief state of the art of fault injection attacks (also called fault attacks) and existing countermeasures, then we define what is a mutant application and its impact on the card security. The section 4 is dedicated to the different countermeasures we have designed. Finally we present an evaluation of our solutions and conclude with the future works for our researches.

2. Fault Attacks

Faults can be induced into a chip using physical perturbations like: a power spike, the heat, a laser, a clock glitch, etc. These errors can generate different versions of a program (a mutant version) by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on.

To prevent a fault attack to happen, we need to know what are its effects on smart cards.

References [5, 15] present taxonomy of fault models in detail. In our case, we choose a subset of these models. We choose the precise byte error model as the most realistic attack model. When an attacker physically injects energy in a memory cell to change its state and depending of the underlying technology, the memory physically takes the value 0x00 or 0xFF. If memory cells are encrypted the physical value becomes random (more precisely a value which depends of the data, its address, and the encryption key). Thus, we assume that an attacker can:

- Make a fault injection at a precise clock cycle (she can target any operation she wants),
- Only set a byte to 0x00 or to 0xFF (bsr for bit set or reset fault type), or change this byte to a random value which cannot been predicted (random fault type),
- Target any memory cells (precise memory cell of a variable or register).

3. Defining a Mutant Application

The mutant generation and detection is a new research field introduced simultaneously by [3, 14] using the concepts of combined attacks (we have already discussed mutant detection in [13]). To define a mutant application, we use an example on the following debit method that belongs to a wallet Java Card applet. Here, the user pin must be validated prior to the debit operation.

Table 1 presents the corresponding bytecode representation. An attacker wants to bypass the pin test. A fault on the cell containing the conditional test bytecode changes the ifeq instruction (byte 0x60) to a nop instruction (byte 0x00). The resulting Java code and bytecode are showed in Table 2.

```
private void debit(APDU apdu) {
    if ( pin.isValidated() ) {
        // make the debit operation
    }else {
        ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

Table 1 - Bytecode Representation Before Attack

Byte	Bytecode representation
00 : 18 01 : 83 00 04 04 : 8B 00 23 07 : 60 00 3B 10 : ... ... 59 : 13 63 01 63 : 8D 00 0D 66 : 7A	00 : aload_0 01 : getfield #4 04 : invokevirtual #18 07: ifeq 59 10 : ... ... 59 : sipush 25345 63 : invokestatic #13 66 : return

```
private void debit(APDU apdu) {
    // make the debit operation
    ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
}
```

Table 2 - Bytecode Representation After Attack

Byte	Bytecode representation
00 : 18 01 : 83 00 04 04 : 8B 00 23 07 : 00 08 : 00 09 : 3B 10 : ... ... 59 : 13 63 01 63 : 8D 00 0D 66 : 7A	00 : aload_0 01 : getfield #4 04 : invokevirtual #18 07 : nop 08 : nop 09 : pop 10 : ... ... 59 : sipush 25345 63 : invokestatic #13 66 : return

If the attack is successful the pin code check is bypassed, the debit operation is made. Then the exception is thrown but the attacker had already achieved his goal. This attack is an example of a dangerous mutant application: *"an application that has been modified by an attack in such a way that the result is correct for the virtual machine interpreter but that doesn't have the same behavior than the original application"*. In this paper, we present several countermeasures to detect when such modifications happen.

4. Software Countermeasures on Smart Card

The software countermeasures are introduced at different stages during the development process and aimed to strengthen the application code against fault injection attacks. Software countermeasures can be classified by their means:

– *Cryptographic countermeasures* aim to get better implementation of cryptographic algorithms like RSA (which is the most frequently used public key algorithm in smart cards), DES, and hash functions (MD5, SHA-1, etc).

– *Applicative countermeasures* target the application code and generally produce application with bigger size (because the code and the data structures for enforcing the security mechanism are embedded with the functional code of the application). Java is an interpreted language (using a virtual machine); therefore, it is slower to execute than a native language (like C or assembler). So this kind of countermeasures suffers of low execution time.

– *System countermeasures* aim to harden the system to offer a safe environment to the applications (using dual computation of sensitive data, execution counter, etc). The main advantage is that the system and the protections are stored in the ROM, which is a less critical resource than the EEPROM and cannot be attacked (thanks to checksum mechanisms that detect modification of data stored in the ROM.

– *Hybrid countermeasures* are at the crossroad between applicative and system countermeasures. The application programmer inserts data in the applications that are exploited later by the system to protect the application code against fault attacks. They provide a better balance between application size and the CPU overheads.

All previous categories (with the exception of the cryptographic countermeasures) use a generalist approach to detect the fault because they don't focus on a particular algorithm. The mechanisms proposed in this paper are hybrid countermeasures.

4.1. General Idea Behind the Proposed Countermeasures

The proposed solution uses Java annotations, which are used by the virtual machine interpreter. Then, the JVCM switches to a "secure mode" when it enters a function which has been annotated as sensitive. The following code fragment shows the use of an annotation on the debit method. The @SensitiveType annotation denotes that this method must be checked for integrity with the FoB mechanism.

```
@SensitiveType{
    sensitivity= SensitiveValue.INTEGRITY,
    proprietaryValue="FoB"
}
private void debit(APDU apdu) {
    if ( pin.isValidated() ) {
        // make the debit operation
    } else {
        ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
    }
}
```

We have developed a tool that processes an annotated classfile. Thus, the annotations become a custom component containing security information. This is possible in JavaCard

because the Java Card specification [20] allows adding custom components to a classfile. In this case, a virtual machine processes custom components if it knows how to use them or else, silently ignores them. To process the information in these custom components the virtual machine must be modified but annotations used in Java code keeps its portability property (a JCVM which doesn't have support for countermeasure annotations can still execute the applet, albeit without the protection offers by our mechanisms).

This approach requires from the attacker to succeed to simultaneously inject two faults at the right time, one on the application code and the other on the system during its interpretation of the code, which is something hard to realize, and outside the scope of the chosen fault model (an attacker can only make one fault at a time).

4.2. First Detection Mechanism: The Field of Bit (FoB)

This countermeasure considers that if an attack modifies an opcode by another one, it is possible that operands of the previous opcode are inconsistent with the new one. More precisely we can obtain the following situations:

1. An increase of the number of operands for the instruction: when add (no operand) is replaced by icmpeq (one operand) for instance;
2. A reduction of the number of operands for the instruction: when aload (one operand) is replaced by athrow (no operand) for instance;
3. The same number of operands: when iload (one operand) is replaced by return (one operand) for instance.

4.2.1. Off-card: After compilation of Java file, a tool generates a field of bit representing the type of each element in the method's byte array. In this field, opcode is marked with an X (execute) and operands representing the data manipulated by this instruction are marked with an R (read). Table 3 shows the byte representation and the corresponding field of bit for the debit method.

Table 3 - Field of Bit

Byte	FoB
00 : 18	X
01 : 83 00 04	XRR
04 : 8B 00 23	XRR
07 : 60 00 3B	XRR
10 :	... ...
59 : 13 63 01	XRR
63 : 8D 00 0D	XRR
66 : 7A	X

The resulting table is saved as a component of the classfile (as the Java Card specification allows the creation of custom component).

4.2.2. On-card: During the method's byte array interpretation, the JVCM interpreter has first to check the kind of security enforced. It checks the presence of the "FoB" annotation parameter. If it is present then it verifies the concordance between the instruction currently interpreted and the values stored in the table. For example, if we assume that we are in the same configuration as in Table 2. We can note that the field of bit has changed its value from Table 4 (before the attack) to Table 5 (after the attack).

Table 4 - Field of Bit Before Attack

FoB	X	X	R	R	X	R	R	X	R	R	…	X	R	R	X	R	R	X
PC	0	1	2	3	4	5	6	7	8	9	…	59	60	61	62	63	64	65

Table 5 - Field of Bit After Attack

FoB	X	X	R	R	X	R	R	X	X	X	…	X	R	R	X	R	R	X
PC	0	1	2	3	4	5	6	7	8	9	…	59	60	61	62	63	64	65

Thus, when the interpreter tries to execute the ifeq at the PC 7, it detects that at PC 8 the table is filled with an X instead of R. In other term, the byte's semantic has changed from readable to executable. Thus, the interpreter tests for each instruction if MethodByteArray[vmpc] is equal to FieldofBit[vmpc]. If it is different, the interpreter stops the execution because a fault has probably occurred.

This method has a drawback: if an instruction replaces another instruction that has the same number of operands then the replacement is not detected; this case is called an indistinguishable replacement. Fortunately, with the chosen fault model, we have estimated that the probability for this to happen in Java Card application is 10% (this percentage has been obtained by statistics over 10000 instructions in 15 different applets).

4.3. Second Detection Mechanism: The Basic Block (BB)

This protection mechanism uses the basic block notion and the application control flow notion (from graph theory) to check the code integrity. A basic block is a sequence of instructions with a single entry point and a single exit point. Thus execution of a basic block can start only at its entry point, and can only leave at its exit point. These points are called leaders. To compute the basic blocks, the main task is to find the set of leaders:

- The first instruction of the method and each first instruction of every handler of method is a leader.
- Each instruction that is the target of an unconditional branch (goto, jsr, ret) is a leader.
- Each instruction that is the target of a conditional branch (ifeq, iflt, ifne, ifgt, ifge, ifnull, ifnotnull, if_icmpeq, if_icmpne, if_icomplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq) is a leader.
- Each instruction target of a composed conditional branch (tableswitch or lookupswitch) is a leader.

Each instruction that immediately follows a conditional or unconditional branch, or a return instruction (ireturn, areturn and return), or a composed conditional branch instruction is a leader.

Each individual leader starts a basic block, consisting of all instructions up to the next leader or to the end of the method's byte array. For simplification of this paper we enclose each method invocation (invokevirtual, invokespecial, invokestatic, and invokeinterface) in a basic block of its own and we do not consider the instruction that can implicitly or explicitly throws an exception.

4.3.1. Off-card: The next step is to compute the checksum of each block using the XOR operation on all the bytes composing a basic block. This computation leads to a table containing for each basic block the following data:

- The PC of its beginning,

– The PC of its ending,
– The value of its checksum.

Then this table is stored in the classfile as a custom component and sent to the card. The interpreter has to be modified to use the previous data. For instance, if we refer to the Table 1, the debit method shows two basic blocks: from 00 to 09 with a checksum value of 0x61, and from 59 to 66 with a check sum value of 0x98.

4.3.2. On-card: During runtime, the virtual machine interpreter first checks the presence of the "BB" annotation parameter that corresponds to the current detection mechanism. If this tag is present, it computes the basic blocks on the fly and compares them with the ones stored in the custom components of the classfile, using the following conditions:

1. The program counter (PC) of the current instruction exists in the table as a beginning or an ending for a basic block.
2. If the first condition is verified then the JVCM checks that the current instruction is really a leader.
3. During the basic block processing, it computes step by step the checksum value until the end of the current basic block.
4. At the end of the current basic block, before resuming to the next instruction it checks that the checksum computed for the current block is equal to the value stored for this block. If they aren't, it's probably because of an attacks and the interpreter can take appropriate measures for instance stops the execution.

If a fault occurs (according to the chosen fault model) then the mechanism will detect it whatever the modification is because the XOR result will also change. If two bytes are modified by a fault the resulting XOR can be different whence the importance of the fault model: an attacker can only modify one byte at a time.

4.4 Third Detection Mechanism: The Path Checking (PC)

This mechanism works on the byte code where all transformations and computations are done on a server (off-card). It is a generalist approach that is not dependent of the type of application. But it cannot be applied to native code such as cryptographic algorithm.

4.4.1 Off-card: The first step is to create the control flow graph of the annotated method, by **splitting** its code into basic blocks (as defined in Section 4.3) and by linking them. Thus after basic block computation our tool computes its control flow graph where the basic blocks represent the vertices of the graph and directed edges denote a jump in the code between two basic blocks (Fig. 2).

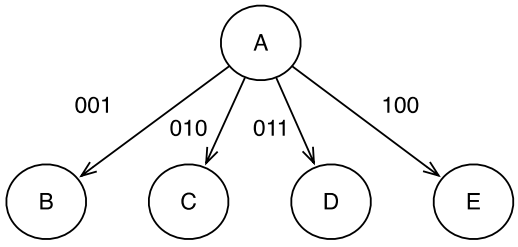


Fig. 1. Coding a Switch Instruction (a Tableswitch or a Lookupswitch) with 3 Branches

The third step is to compute for each vertex a list of paths from the beginning vertex. The computed paths are encoded using the following convention:

- Each path begins with the tag "01". This is to avoid an attack that changes the first element of a path to 0x00 or to 0xFF.
- If the instruction that ends the current block is an unconditional or conditional branch instruction, when jumping to the target of this instruction (represented by a low edge in Fig. 2), then the tag "0" is used.
- If the execution continues at the instruction that immediately follows the final instruction of the current block (represented by a top edge in Fig. 2), then the tag "1" is used.

If the final instruction of the current basic block is a switch instruction, a particular tag is used, composed by the number of bits necessary to encode all the targets. For example, if we have four targets, we use three bits to code each branch (like in Fig. 1). Switch instructions are not so frequent in Java Card applications. And to avoid a great increase of the application size that uses this countermeasure, they should be avoided.

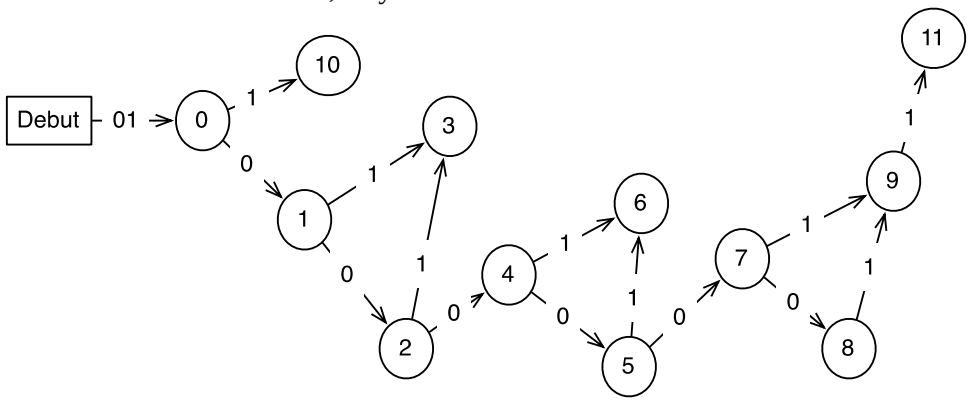


Fig. 2. Control Flow Graph of the Debit Method

Thus a path from the beginning to a basic block is X0...Xn (where X corresponds to a 0 or to a 1 and n is the maximum number of bits necessary to code the path). In our example, to reach the basic block 9, which contains the update of the balance amount, the paths are: 01 0 0 0 0 0 0 1 and 01 0 0 0 0 0 1.

4.4.2. On-card: When interpreting the byte code of the method to protect, the virtual machine detects the annotation and analyzes the type of required security mechanism. In our case, it is the path check security mechanism. Then during the code interpretation, it computes the execution path. For example, when it encounters a branch instruction, when jumping to the target of this instruction then it saves the tag "0", and when jumping to the instruction that follows it saves the tag "1". Then prior to the execution of a basic block, it checks that the followed path is an authorized path i.e. a path that belongs to the list of path computed for this basic block. For the basic block 9, it is necessary one of the two previous paths, if not it is probably because to reach this point the interpreter has followed a wrong path; therefore, the card can lock itself.

In the case a loop is detected (backward jump) during the code interpretation, the interpreter checks the path for the loop, the number of references and the number of values on the operand stack before and after the loop, to be sure that for each round the path remains the same.

5. Evaluations

This section aims to evaluate two aspects of the detection mechanisms presented in this paper. The first aspect is their efficiency in detecting the fault attacks according to the chosen fault model. The second aspect is their efficiency in term of resources consumption, because smart cards have small memory footprints and computation power with a high challenge/response latency constraint. Thus, these two categories of metrics are needed to evaluate the adequacy of the countermeasures in a production environment.

5.1. Runtime Overhead

The first category of metrics is the memory footprint and the CPU overhead of our countermeasures. They have been obtained using the SimpleRTJ [1] a Java virtual machine that targets highly restricted constraints device like smart card. The hardware platform used for the evaluation is an AT91EB40A evaluation board with an ARM7TDMI microprocessor. The internal clock speed of this board if by default 32,768 MHZ, which is the same as in a high-end smart card. For the memory, it has 2Mb of flash and 256 Kb of RAM and can use a 16 or 32 bits memory addressing. These characteristics make this hardware as close as possible of common smart cards.

SimpleRTJ is a Java virtual machine designed for small memory footprints embedded devices based on 8, 16, or 32 bits microprocessor. It uses off-card library linking process like java card 2.x. It embeds the entire API to optimize execution and memory overhead. The source code for SimpleRTJ already exist for the evaluation board, so we have modified the code relating to the interpreter to integrate our countermeasures (data type for handling custom components and code fragments to detect modification caused by fault attacks).

Table 6 shows the metrics for resources consumption obtained by using the detection mechanisms on all the methods of our test applications. The increase of the application size is variable for the basic block detection mechanism and the path checking detection mechanism. This is due to the number of paths and to the number of basic blocks generated off card. Even if the path checking mechanism is close to 10 % and the basic block to 5 % of application size and respectively of 8 % and 5 % of CPU overhead, the developer can choose when to activate only for sensitive methods to preserve resources thanks to the annotation mechanism. The field of bits mechanism is very light in comparison of the two other mechanisms, and the memory overhead is constant because the data generated by this mechanism depend only on the number of bytes of an application. These countermeasures need small changes on the virtual machine interpreter. So we can conclude that they are affordable countermeasure for the aimed target.

Table 6 - Runtime Overhead

Countermeasures	EEPROM	ROM	CPU
FoB	+3 %	+1 %	+3 %
BB	+5 %	+1 %	+5 %
PC	+10 %	+1 %	+8 %

5.2. Detection Efficiency

To evaluate the detection mechanisms, we have developed an abstract Java Card virtual machine interpreter. This abstract interpreter is designed to follow a method call graph, and

for each method of a given Java Card applet, it simulates a Java Card method's frame. A frame is a memory area allocated for the operand stack and the local variables of a given method.

The interpreter can also simulate an attack by modifying the method's byte array. This is important because it allows reproducing faults on demand. On top of the abstract interpreter, we have developed a mutant generator. This tool can generate all the mutants corresponding to a given application according to the chosen fault model. To do that, the mutant generator changes each opcode value from 0x00 to 0xFF, and for each of these values an abstract interpretation is made. If the abstract interpretation does not detect a modification then a mutant is successfully created. We have also developed a program to reverse engineering mutant application and thus understand the corresponding Java source.

The mutant generator has different modes of execution:

- The basic mode: the interpreter executes the instruction pushing and popping element on the operands stack and uses local variables without check. In this configuration instructions can use elements of other methods frame like using their operand stacks or using their locals. When running this mode, it has no countermeasures activated.
- The simple mode: that correspond to a partial BCV (ByteCode Verifier) implementation. The interpreter checks that no overflow or no underflow occurs, that the used locals are inside the current table of locals, and that when a jump occurs it's done inside the method. They consist in some of the verifications done by the Java BCV.
- The advanced mode: it is the simple mode with the ability to activate or to deactivate a given countermeasures like the developed ones: path checking mechanism (PC), field of bits mechanism (FoB), or the basic block mechanism (BB).

5.2.1. Detection coverage: The Table 7 shows the reduction of generated mutants in each mode of the mutant generator for an application. The second line shows mutants generated in each mode of the mutant generator.

Table 7 – Non Detected Mutants by Countermeasures

Applications	Basic mode	Partial BCV	FoB	BB	PC
Wallet	440	434	18	0	37
Utilities	2740	2226	157	0	230
TeaApplet	1944	1916	95	0	163

The most efficient countermeasure is the basic block, which detects all the mutants that have been generated according to the fault model. Because this technique use a checksum value that is calculated with all the byte that compose a basic block.

The field of bit detection mechanism fails to detect mutants that have been generated by indistinguishable instruction replacement. For the evaluation purposes we have generated all the potential mutants, but otherwise we have seen that the indistinguishable replacements have a low probability to appear (see section 4.2.2).

The path checking fails to detect mutant when the fault that generates the mutant don't influence the control flow of the code. Otherwise, when a fault occurs that alter the control flow of the application then this countermeasure detects it. With this countermeasure, it becomes impossible to bypass systems calls like cryptographic keys verification.

5.2.2. Latency: The latency is the number of instructions executed between the attack and the detection. In the basic mode no latency is recorded because no detection is made. This value is also really important because if a latency if too high maybe instructions that modify persistent memory like: putfield, putstatic or an invoke instruction (invokestatic, invokevirtual, invokespecial, invokeinterface) can be executed. This can cause a modification of sensitive data stored in the smart card (like cryptogrtaphic keys, pin code, etc). So this value has to be as small as possible to lower the chances to have instructions that can modify persistent memory. The Table 8 shows the average latency obtained on the previous applications.

Table 8 - Average Latency by Countermeasures in Number of Instructions

Applications	Basic mode	Partial BCV	FoB	BB	PC
Wallet	-	2	2,42	2,72	3
Utilities	-	106,77	8,61	10,53	13,06
TeaApplet	-	56,86	5,82	7,66	11,72

We can see that the average latency of the developed countermeasures is lower than the Partial BCV countermeasures. Thus, they are more efficient than the common protection that we can find in a smart card.

6. Future Works: Compression Detection Mechanism

We are now working on a detection mechanism that is based on bytecode compression. The idea behind this technique comes from authors of [4,6]. The goal of this mechanism is to reduce the Java Card instruction set because the less instruction we have, the less are the probability for an attack to make a replacement that makes sense for the virtual machine interpreter (according to our fault model). In fact, the fault model induces that when an attack is a success, then the probability of a change to happen is 1/255. In the Java Card specification, there are 185 instructions, all of them are useful to represent a Java Card application, so the probability that a change makes sense for a Java Card interpreter is given by Equation 1. With this equation we can understand that the lower the size of the instruction set is, the lower the probability will be.

$$P = \frac{1}{255} * SizeOfInstructionSet = \frac{185}{255} \approx 0{,}72$$

Equation 1

To achieve this goal, we have noticed that in Java Card's application some opcode instructions have a high probability to come along. These instructions form what we call "patterns" A pattern is a group of n instructions that always come together in an application. From a pattern we generate a new instruction, for example, the pattern "sipush 26368, invokestatic 13" becomes the new instruction " sipushandinvokevirtual 26368 13". When all the patterns have been found we obtain a new instruction set that is smaller than the original instruction set.

The principle is theoretically to apply a compression algorithm to the Java Card binary file and to modify the virtual machine to integrate the new instruction set. This mechanism has some advantages first the binary will be smaller than regular one and according to [4] the execution time of the application can be reduced.

7. Conclusions

We had presented in this paper some new approach that are affordable for smart card and that are fully compliant with the Java Card 2.x and 3.x specification. Moreover, it does not have high computation constraint and the produced binary files are under a reasonable limit in term of size. It also does not disturb the applet conception workflow. It saves time to the developer who wants to produce secured applications thanks to the use of the sensitive annotations. Finally, it needs small modifications of the java virtual machine. It also has a good mutant detection capacity.

We have implemented all these countermeasures inside a smart card to have metrics concerning memory footprint and processor overhead, which are all affordable for smart card. In this paper, we presented the second part of this characterization to evaluate the efficiency of countermeasures in smart card operating system. We provide a framework to detect mutant applications according to a fault model and a memory model. This framework can provide to a security evaluator officer all the source code of the potential mutants of the application. She can decide if there is a threat with some mutants and then to implement a specific countermeasure. Within this tool, either the developer, or security evaluator officer can take adequate decision concerning the security of its smart card application. For the developer company, reducing the size of the embedded code minimizes the cost of the application. For the security evaluator, it provides a semi automatic tool to perform vulnerability analysis.

References

[1] RTJ Computing, 2010. url: http://www.rtjcom.com/main.php?p=home.

[2] C. Aumuller et al. “Fault attacks on RSA with CRT: Concrete results and practical countermeasures”. In: Lecture Notes in Computer Science (2003), pp. 260–275.

[3] G. Barbu, H. Thiebeauld, and V. Guerin. “Attacks on Java Card 3.0 Combining Fault and Logical Attacks”. In: Smart Card Research and Advanced Application, Cardis 2010 LNCS 6035 (2010), pp. 148–163.

[4] G. Bizzotto and G. Grimaud. “Practical Java Card bytecode compression”. In: Proceedings of RENPAR14/ASF/SYMPA. Citeseer. 2002.

[5] J. Blomer, M. Otto, and J.P. Seifert. “A new CRT-RSA algorithm secure against Bellcore attacks”. In: Proceedings of the 10th ACM conference on Computer and communications security. ACM New York, NY, USA. 2003, pp. 311–320.

[6] L.R. Clausen et al. “Java bytecode compression for low-end embedded systems”. In: ACM Transactions on Programming Languages and Systems (TOPLAS) 22.3 (2000), p. 489.

[7] Global platform group. Global platform official site. 2010. url:http://www.globalplatform.org.

[8] L. Hemme. “A differential fault attack against early rounds of (triple) DES”. In: Cryptographic Hardware and Embedded Systems-CHES 2004 (2004), pp. 170–217.

[9] J. Iguchi-Cartigny and J.L. Lanet. “Developing a Trojan applets in a smart card”. In: Journal in Computer Virology (2009), pp. 1–9.

[10] Sun Mycrosystems. Java CardTM 3.0.1 Specification. Sun Microsystems. 2009.

[11] G. Piret and J.J. Quisquater. “A differential fault attack technique against SPN structures, with application to the AES and Khazad”. In: Cryptographic Hardware and Embedded Systems-CHES 2003 (2003), pp. 77–88.

[12] A.A. Sere, J. Iguchi-Cartigny, and J.L. Lanet. “Automatic detection of fault attack and countermeasures”. In: Proceedings of the 4th workshop on Embedded Systems Security. ACM. 2009, pp. 1–7.

13] A.A. Sere, J. Iguchi-Cartigny, and J.L. Lanet. “Mutant applications in smart card”. In: Proceedings of CIS 2010 (2010).

[14] E. Vetillard and A. Ferrari. “Combined Attacks and Countermeasures”. In: Smart Card Research and Advanced Application, Cardis 2010 LNCS 6035 (2010), pp. 133–147.

[15] D. Wagner. “Cryptanalysis of a provably secure CRT-RSA algorithm”. In: Proceedings of the 11th ACM conference on Computer and communications security. ACM New York, NY, USA. 2004, pp. 92–97.

J Comput Virol (2010) 6:343–351
DOI 10.1007/s11416-009-0135-3

ORIGINAL PAPER

Developing a Trojan applets in a smart card

Julien Iguchi-Cartigny · Jean-Louis Lanet

Received: 4 January 2009 / Accepted: 26 August 2009 / Published online: 11 September 2009

Abstract This paper presents a method to inject a mutable Java Card applet into a smart card. This code can on demand parse the memory in order to search for a given pattern and eliminate it. One of these key features is to bypass security checks or retrieve secret data from other applets. We evaluate the countermeasures against this attack and we show how some of them can be circumvented and we propose to combine this attack with others already known.

1 Introduction

A smart card is a piece of plastic, the size of a credit card, in which a single chip microcontroller is embedded. Usually, microcontrollers for cards contain a microprocessor and different memories: Ram (for run-time data and OS stacks), Rom (in which the operating system and the "romized" applications are stored), and Eeprom (in which the persistent data are stored). Due to strong size constraints on the chip, the amount of memory is small. Most smart cards sold today have at most 5 KB of Ram, 256 KB of Rom, and 256 KB of Eeprom. This chip usually also contains some sensors (like light sensors, heat sensors, voltage sensors, etc.), which are used to disable the card when it is physically attacked.

A smart card can be viewed as a secure data container, since it securely stores data and it is used securely during short transactions. Its safety relies on the underlying hardware. A physical attack is quite difficult because the chip in a card is embedded with sensors covered with a resin, and all components are on the same chip (difficult to probe an internal bus). The software is the second barrier for its safety. The embedded programs are usually designed neither for returning nor modifying sensitive information without being sure that the operation is authorized. Java Card is a kind of smart card that implements the standard Java Card 3.0 [1] in one of the two editions "*Classic Edition*" or "*Connected Edition*". Such a smart card embeds a virtual machine which interprets codes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol defined by Global Platform [2]. This protocol ensures that the owner of the code has the necessary credentials to perform the action.

Java Card is an open platform for smart cards, i.e. able of loading and executing new applications after issuance. Thus, different applications from different providers run in the same smart card. Thanks to type verification, the bytecodes delivered by the Java compiler and the converter (in charge of delivering compact representation of class files) are safe, i.e. the application loaded is not hostile to other applications in the Java Card. Furthermore, the Java Card firewall checks permissions between applications in the card, enforcing isolation between applications. Until now, it was safe to presume that the firewall was efficient enough to avoid bad behaviour from malicious applications (crafted applet modified after off-card verification). In this paper, we will show that an attacker can generate malicious applications which bypass firewall restrictions and modify other applications, even if they don't belong to the same security package. Since the firewall is the only mandatory on-card security mechanism in the Java Card our work reveals an important security issue. Java card has been an important improvement in the smart card world due to the security aspect of the Java platform. Nevertheless some attacks have been successful in retrieving secret data from the card. Thus we will present in

J. Iguchi-Cartigny · J.-L. Lanet ( )
XLIM/DMI/SSD, 83 rue d'Isle, 87000 Limoges, France
e-mail: jean-louis.lanet@xlim.fr

this paper a methodology to get access to data, which should bypass Java security components.

In the next section, we describe the different components involved in the Java Card's platform security. Section 3 describes the state-of-the-art of smart cards logical attacks. Then we introduce, a methodology to implement a Trojan in the card in Sect. 4. Section 5 presents the evaluation of our attack and encountered countermeasures. Finally, the last section presents our future works and concludes.

2 Java Card

Java Card is quite similar to any other Java edition, it only differs (at least for the Classic Edition) from standard Java in three aspects: (i) restriction of the language, (ii) runtime environment and (iii) the applet life cycle. Due to resource constraints the virtual machine in the Classic Edition must be split into two parts: the bytecode verifier executed off-card, is invoked by a converter while the interpreter, the API and the Java Card runtime environment (JCRE) are executed on board. The bytecode verifier is the offensive security process of the Java Card. It performs the static code verifications required by the virtual machine specification. The verifier guarantees the validity of the code being loaded in the card. The bytecode converter transforms the Java class files, which have been verified and validated, into a format that is more suitable for smart cards, the CAP file format. An on-card loader installs the classes into the card memory. The conversion and the loading steps are not executed consecutively (a lot of time can separate them). Thus, it may be possible to corrupt the CAP file, intentionally or not, during the transfer. In order to avoid it, the Global Platform Security Domain checks the integrity and authenticates the package before its registration in the card. Along this paper when talking about Java Card we will refer to the "*Classic Edition*".

2.1 The Java Card platform

Developers write Java applets which can be uploaded to Java Card smart cards. Then, the applet's bytecode is interpreted by the embedded Java Card virtual machine (JVCM), an implementation of the Java Card Platform Specification. Such smart cards are thus open platforms, where new applications (or applets) can be uploaded after issuance of the card to the final user.

Due to resource constraints, the JCVCM must be split into two parts:

- First, the bytecode verifier (BCV) and the converter are executed outside the card (off-card) to generate a valid CAP file.
- Second, the interpreter, the API and the Java Card runtime environment (JCRE) execute and handle the applet behaviour inside the smart card.

In the first part, the bytecode verifier acts as an offensive security process located in the JCVM. It performs static code analysis on the Java class files, which is required by the JVM specification. Then, the bytecode converter transforms these files into a more suitable format used with smart cards: a CAP file. This file is a JAR file containing a compact representation of one or several class files adapted to the smart card constraints.

The next step is the storage of the Java classes in the card memory by the on card loader. During the loading process, the CAP file is first unpacked into individual components, which are then downloaded into the card sequentially, component by component by the off-card loader. A special application on the card—the Installer–receives the newly downloaded applet and stores its content into the persistent memory. It also requests from the system the linking of the new classes to the packages already present on the card. After loading and linking, the package is ready to be executed by the JCVM.

2.2 Java Card security

The Java Card platform is a multi-application environment in which an applet's critical data must be protected against malicious access from other applets [3]. To enforce protection between applets, traditional Java technology uses type verification, class loaders and security managers to create private namespaces for applets. In a smart card, it is not possible to comply with the traditional enforcement process.

Firstly, the type verification is executed outside the card due to memory constraints. Secondly, class loaders and security managers are replaced by the Java Card firewall.

2.3 The bytecode verifier

Allowing code to be loaded into the card after post-issuance raises the same issues as with web applets. An applet that has not been compiled by a compiler (handmade bytecode) or that has been modified after compilation can break the Java sandbox model. Thus the client must check that the Java typing rules are preserved at the bytecode level.

The Java language is strongly typed, which means that every variable and every expression has a type that is determined at compiling time. Type mismatches in the source code are detected at compile time as well, and Java bytecode is also strongly typed. Still, local and stack variables of the virtual machine do not have unchanging fixed types even in the scope of one method execution. Not all type mismatches are detected at runtime, and this allows building malicious

applets exploiting this issue. For example, pointers are not supported by the Java programming language. Though, they are extensively used in Java Virtual Machine, where they are referred to as references. Thus, the absence of pointers reduces the number of programming errors. But it does not stop attempts to break security protections by disloyal use of pointers.

Bytecode verifier (BCV) is a crucial security component in the Java sandbox model: any bug in the verifier causing an ill-typed applet to be accepted can potentially enable a security attack. At the same time, bytecode verification is a complex process involving elaborate program analyses. Moreover such an algorithm is very costly in terms of time consumption and memory usage. For these reasons, many cards do not implement such a component and rely on the fact that it is the responsibility of the organisation that signs the code of the applet to ensure that the code is correctly typed.

2.4 Java card firewall

The separation between different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of contexts. When an applet is created, the JCRE uses a unique applet identifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes coming from the same Java Card package, they are considered as belonging to the same context. There is a superuser context, called the JCRE context. Applets belonging to this context can access objects from any other context on the card.

Every object is assigned to a unique owner context which is the context of the applet that created the object. A method of an object is said to be executed in the owner context of the object. It is this context that decides whether access to another object is allowed or not. The firewall isolates the contexts in such a way that a method executing in one context cannot access any attribute or method of objects belonging to another context.

There are two ways to bypass the firewall: via JCRE entry points and via shareable objects. JCRE entry points are objects owned by the JCRE that have been specifically designated as objects that can be accessed from any context. The most significant example is the APDU buffer in which commands sent to the card are stored. This object is managed by the JCRE and, in order to allow applets to access this object, it is designated as an entry point. Other examples include the elements of the table containing the AIDs of the applets installed on the card. Entry points can be marked as temporary. References to temporary entry points cannot be stored in objects (this is enforced by the firewall).

2.5 The sharing mechanism

To support cooperative applications on a single card, the Java Card technology provides well defined sharing mechanisms. The shareable interface object (SIO) mechanism is the mechanism in the Java Card platform intended for applets collaboration. The `javacard.framework` package provides a tagging interface called Shareable, and any interface which extends the Shareable interface will be considered as a Shareable interface. Requests for services to objects implementing a Shareable interface are allowed by the firewall mechanism. When a server applet wants to provide services to other applets within the Java Card, it must define the services it wants to export in an interface tagged as Shareable.

Within the Java Card, only instances of classes are owned by applets (i.e. are within the same security context), classes themselves are not. No runtime check is performed when a static field is accessed or when a static operation is invoked. This means that static fields and operations are accessible from any applet; however, objects stored in static fields belong to the applet which instantiates them. The server applet may decide whether to publish its Shareable Interface Objects (SIOs) in static fields, or return them in static operations.

3 Related work

In order to retrieve some data from the card, we can use two different approaches: physical attacks or logical attacks.

3.1 Physical attack

A first class of physical attacks is side-channel attacks [4]. These non-invasive attacks consist in observing an unintended physical effect of computation (timing, data exchanged on the I/O channels, power consumption, electromagnetic noise, etc.) to discover information like the secret key used in some cryptographic operations. For instance, SPA and DPA stand for Simple and Differential Power Analysis, respectively, and aim at exploiting the information leaked through characteristic variations in the power consumption of electronic components. Fault induction attack, or perturbation attack, consists in changing the behaviour of the component in order to create an exploitable error [5]. Such faults can be induced using different means, including glitches (a surge of power on one of the cards I/O ports), light/laser exposure, etc. The attack will typically try to make cryptographic operations weaker (by creating faults that can be used to recover key or plaintext information), or avoid or corrupt the results of checks (such as authentication or lifecycle state checks, or else change the program flow).

3.2 Logical attack

As explained in [6] logical attacks consist in executing ill-formed applications, i.e., malicious applications that are made of illegal bytecode instructions sequences or that do not have valid bytecode parameters. Such attacks are limited to cards for which:

- post issuance is allowed,
- the attacker has the credentials (Security Domain keys),
- the card must not include a bytecode verifier.

W. Mostowski et E. Poll [7] proposed several attacks on Java Card using an ill-typed code. One way is to exploit type confusion between primitive arrays of different types. By convincing the applet to handle a byte array as a short array, it would theoretically be possible to read or write twice the size of the original byte array. They use different approaches: a flaw in the implementation of the transaction mechanism and the shareable interface.

Another way [8,9] is to trick the virtual machine to handle an object as an array. Hence, fields from a forged object can be seen as length of the array if they are stored at the same offset in the physical memory. The attacker would be able to set the size of the array and thus have access to the whole memory of the Java Card. The authors exploit the previous attack to handle reference as short (and short as reference), thus allowing reading and writing existing references, something theoretically impossible on Java Card. They proposed several exploitations of this method. First, it is possible to swap the references of two objects even if they have incompatible types. An attacker can also manipulate the system-owned applet identifier (AID) and thus impersonate a valid applet (using a stolen AID). Finally, it would be possible to spoof references and thus be able to read part of the memory. This last step was not conclusive, as most of the cards refuse spoofed references.

Hyppönen [10] proposed a way to exploit weaknesses in static instructions (*getstatic* and *putstatic*) and in the reference location component of the CAP file which can lead to reference spoofing. The *getstatic* instruction is used to get a static field from a class. The two operands of this opcode are used to build an index in the constant pool. During applet loading, the on-card linking process will replace the two operands with an address in real memory. The idea of the attack is to remove the entry in the Reference Location component so that the operand of the *getstatic* will not be resolved. Thus it becomes possible to assign any value to the two operands, and therefore to point to any real address. This attack works (in the absence of a bytecode verifier) because no context check is done during access to static field. However, the author did not present any experimental results or an implementation of the attack.

In this paper, we will prove that such an attack is possible, and we propose a very efficient implementation using another bytecode weakness that allows us to generate a safe mutable code.

4 The malicious code

Instead of dumping the memory byte after byte we use the ability to invoke an array that can be filled with any arbitrary bytecode. Within this approach, we are able to define a search and replace function. To demonstrate the application of such attack, please consider the following generic code, often used to check if a PIN code has been validated.

```
public void debit (APDU apdu )
{
...
if (!pin.isValidated())
{
ISOException.throwIt(SW_AUTH_FAILED)
}
// do safely something authorized
}
```

The pin object is an instance of class *ownerPin*, which is a secure implementation of a PIN code (ratification counter decrements before check and so on). If the user sets a wrong PIN code, an exception is thrown. The goal of our Trojan is to search for the bytecode of this exception treatment and to replace it with a predefined code fragment. For example, if the Trojan finds in memory the pattern 11 69 85 8D 00 12 and if the owner of this method is the targeted applet then the Trojan replaces it by the following pattern: 00 00 00 00 00 00. Knowing that the bytecode 00 stands for the NOP instruction, the original code becomes:

```
public void debit (APDU apdu )
{
...
if (!pin.isValidated())
{ }
// do safely something authorized
}
```

The interest of the search and replace Trojan is obvious. Of course if the Trojan is able to perform such an attack it could also scan the whole memory and characterize the object representation of the virtual machine embedded into the card. It becomes also possible to get access to the implementation of the cryptographic algorithms which in turn can be exploited to generate new attacks.

The basic hypotheses of this attack are: the attacker has the credentials to download his own code into the Java Card and the card does not include a bytecode verifier.

4.1 The mutable applet

The first step is to get a reference to an array located in the context of one of our own applets. Suppose that the following function is in our applet:

```
public short getMyAddresstabByte(byte[] tab)
{
short dummyRef=(byte)0x55AA;
tab[0] = (byte)0xFF; // second instruction
return dummyRef;
}
```

A look at the bytecode level shows that the second instruction is an aload 1, so the reference of the array tab is on the top of the stack after the second instruction. If we change each bytecode by the instruction NOP between the second instruction and the return statement, then the function will return the reference of the array tab instead of the short *dummyRef*. Thus, the array reference can be sent back to the terminal (as a short) using an APDU command. Using this bytecode manipulation of the external file we are able to get a valid reference on elements belonging to our security context.

We begin by defining an array variable (called codeD). The compiled code is the following:

```
public byte[] codeD = {(byte)0x01, (byte)0x00, (byte)0x7D,
(byte)0x00, (byte)0x00, (byte)0x78};
```

If you consider the contents of this array as a method, we have now a function dedicated to read a static field. The two first bytes correspond to the header of the method and will never be interpreted as bytecode.

```
//flags : 0
//max_stack : 1
//nargs : 0
//max_locals : 0
00 aconst_null // header of the method
01 nop // i d e m
02 getstatic_s 0 0
05 sreturn
We can modify the value of the third and fourth bytes with
an APDU command,
and thus choose the operands of getstatic s:
codeD[3]= apduBuf[ISO7816.OFFSET_CDATA];
codeD[4]= apduBuf[ISO7816.OFFSET_CDATA+1];
```

We need a dummy definition (*functionToReplace()*) to generate a reference on a static method:

```
// function to replace
static public short functionToReplace()
{
return ad;
}
```

Now, we can write the loop used to search and replace:

```
For (i=0...){
// to generate a ref to be replaced later
Util.setShort(searchBuf,k,functionToReplace());
codeDump[4]++; //increment low address
if (codeDump[4] == (byte)0x00)
{
codeDump[3]++; // increment high address
}
// search and replace the pattern in searchBuf
}
```

We use another weakness on *invokestatic* and we modify the *Method Component* in order to reference the array CodeD instead of the original method. We also have to modify the Reference Location Component of the CAP file to remove the entry as shown in the Fig. 1.

At offset 0x014e in the Method Component we find the *invokestatic* instruction calling *fonctionToReplace()*. We can see that the operand is an index referring to the *Constant Pool Component* and that the operand offset is referenced in the *Reference Location Component*. First, we edit the file and change the value of this operand offset (0x014f) by the value of its successor in the Reference Location Component. Thus the linker will resolve twice the next entry (which refers to the method *setShort()*).

In the second step, we replace the operands of *invokestatic* by the address of the array's content and then we download and install the mutable applet. Note that we need to retrieve the resolved address of this array, which leads to a two step installation. First, during the installation, we send an APDU command to retrieve the address of the array by using the function described in the Sect. 4.1. In the second step, we replace the operand of *getstatic,* denoted s, by the retrieved address, we remove the entry in the reference location component and then we download and install the mutable applet. The Trojan applet is then ready to be initialized by APDU commands at any time during the card live. If the assets to be obtained are present into the card, initialisation command and search and replace command can be process.

Then the invokestatic will invoke the buffer code and we will just have to increment the adequate location in the buffer to parse the whole memory. As soon as we detect the first expected bytecode, we start storing the dumped memory in *searchBuf[k]*. Then we check if the next bytecode is the one expected. If not we reset the index k, and we continue until we have parsed the entire memory. In fact, the Trojan relies on the know-how of the internal representation of the objects in the memory. Due to the fact that this information is not public, we had to get the complete internal structure of the CAP file inside the card. Thus, the search and replace method takes this information into account to locate the expected applet referenced by its AID. It also contains the methods used to get the address of the array, as well as a method to setup the initial value of the array.

The greatest difficulty is to modify correctly the CAP file while keeping all the interdependent information correctly. We do not have a tool performing this in an automatic way (which would be very helpful to handle efficiently such attacks).

5 Evaluation of the attack

The Java Cards that have been considered in this paper are publicly available at some web store. We evaluated some cards from six smart card providers and we will refer to the different cards using the reference to the manufacturer

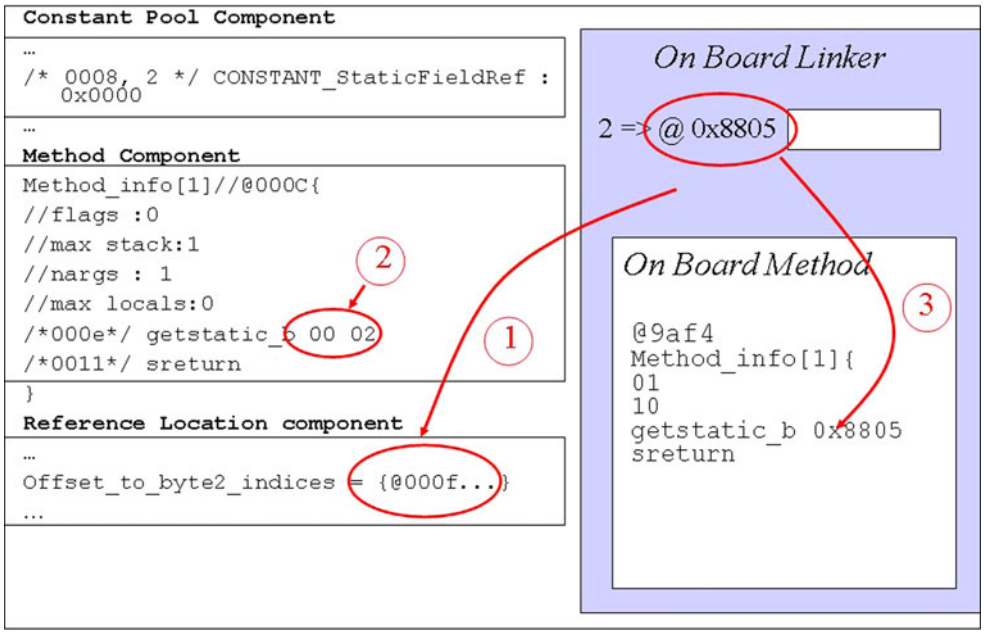


Fig. 1 Embedded linking process

Table 1 Smart Card characteristics used in this study

Reference	Java Card	GP	Characteristics
a-21a	2.1.1	2.01	
a-21b	2.1.1	2.0.1	Same as a-21a plus RSA
a-22a	2.2	2.1	64 KB Eeprom, SLE66CX322 (8051 derivative), page size 1 KB
a-22b	2.1.1	2.0.1	32 KB Eeprom, RSA
b-21a	2.1.1	2.1.2	16 KB Eeprom, RSA P8WE5017 (core 8051)
b-22a	2.1.1	2.0.1	16 KB Eeprom, hW DES
b-22b	2.1.1	2.1.1	P5CD036 (core 8051), page size 2 KB
c-22a	2.1.1	2.0.1	RSA P8WE5033 (core 8051)
c-22b	2.2	2.1.1	64 KB Eeprom, dual interface, RSA
d-22	2.2	2.0.1	
e-21	2.1.1	2.0.1	16 KB Eeprom, SLE66CX165P (8051 derivative), page size 1 KB

associated to the version of the Java Card specifications. At the time we perform this study no Java Card 3.0 were available and the most recent cards we had were Java Card 2.2.

- Manufacturer A, cards a-21a, a-21b, a-22a and a-22b. The a-22a is a USIM card for the 3G, the a-21b is an extension of a-21a supporting RSA algorithm, and the a-22b is a dual interface card.
- Manufacturer B, cards b-21a, b-22a, b22b. The b-21 supports RSA algorithm, the b-22b is a dual interface smart card.
- Manufacturer C, cards c-22a, c22b. The first one is a dual interface card, and the second support RSA algorithm.
- Manufacturer D, card d-22.
- Manufacturer E, cards e-22.

The cards have been renamed with respect to the standard they support. The following table summarizes it (Table 1).

We have conducted this attack on some publicly available evaluation smart cards. While some of these cards implemented countermeasures against this attack, we managed to easily circumvent a few of them.

5.1 Loading time countermeasures

The loader-linker can detect basic modifications of the cap file. Some cards can block themselves when erasing an entry in the `refLocation` component without calculating the offset of the next entry. For instance, the card `a-21a` blocked when detecting a null offset in the `refLocation` component. But it is easy to bypass this simple countermeasure

Table 2 Load time countermeasures

Card reference	RefLocation correct	Type verification
a-21b	x	
c-22b, e-21		x

with elaborate tool able to perform more complex Cap file modifications.

At least three of the evaluated cards have a sort of type verification algorithm (a complex type inference). They can detect ill-formed bytecodes, returning a reference instead of a short for instance. Looking at Common Criteria evaluation reports, it is evident that these cards were out of our hypotheses: they include a bytecode verifier or, at least, a reduced version of it. Thus, such cards can be considered as the most secure, because once the CAP file is detected as ill-formed, cards can reject the CAP file or become mute (for instance `c-22b`) (Table 2).

5.2 Runtime countermeasures

For the remaining cards which pass the loading phase, we can evaluate the different countermeasures done by the interpreter.

A countermeasure consists in checking writing operations. For instance, when writing to an unauthorized memory area (outside the area dedicated to class storage) the card can be blocked or return an error status word. More generally, the cards can detect illegal memory access depending of the accessed object or the bytecode operation. For instance, one card (`c-22a`) limits the possibility to read arbitrary memory location to seven consecutive memory addresses (Table 3).

On the remaining cards, we were able to access and completely dump the memory. The following table summarises the different results we obtained. For each evaluated card, we explain what we have reached with the attack, and then the level of the countermeasure and the portion of the memory dumped (Table 4).

We can compare the countermeasures encountered in this attack with those described in [7]. The first countermeasure described consists in dynamic type inference, i.e. a defensive virtual machine. We never found such a countermeasure on the card we evaluated but may be it is integrated on cards like `c-22b` or `e-21` for which we did not succeed with our attack. Due to the fact that our attack does not modify the array size, any countermeasure trying to detect a logical or a physical size modification will not be efficient. The last countermeasure described concerns the firewall checks. The authors do not try to bypass the firewall using this methodology, thus they did not succeed in discovering this weakness. Nevertheless, their approach could be used, and in particular the buffer overflow for the card `c-22a` for which our attack did not succeed. But if we modify the size of the array, we will be able to bypass the countermeasure on bound check.

Table 3 Runtime countermeasures

Card reference	Memory area check	Memory management	Read access
a-22a		x	x
b-22b	x		
c-22a			x

5.3 Evaluation of other attacks

One of our hypotheses is that the card does not embed a type verifier. In order to relax this hypothesis we evaluate the approach described in [7]. Poll et al. presented a quick overview of classics attacks available on smart card and gave some countermeasures. We will firstly present the different kinds of attacks and after explain which ways we followed.

There are different methods presented in this paper:

- Shareable interfaces mechanisms abuse
- Transaction Mechanisms abuse

The idea to abuse shareable interfaces is really interesting and can lead tricking the virtual machine. The main goal is to have type confusion without the need to edit CAP files. To do that, we have to create two applets which will communicate using the shareable interface mechanism. To create type confusion, each of the applets will use a different type of array to exchange data. During compilation or on load, there is no way for the BCV to detect such a problem.

The problem seems to be that every card they tried, with an on card BCV, refused to allow applets using shareable interface. As it is impossible for an on card BCV to detect this kind of anomaly, Erik Poll emitted the hypothesis that they decided to forbid any use of shareable interface on card with an on card BCV. In our experiments, we succeed to pass a byte array as a short array in all case but when we exceeded the standard ending of the array, an error was checked by the card. This means that the type confusing is possible but a runtime countermeasure is implemented again this attack (Table 5).

The second option was the transaction mechanism. The purpose of transaction is to make a group of operation becomes atomic. Of course, it is a widely used concept, like in databases, but still hard to implement. By definition, the roll-back mechanism should also de allocate any objects allocated during an aborted transaction, and reset references to such objects to null. However, they find some strange cases where the cards keep the reference of objects allocated during transaction even after a roll back. If we can get the

Table 4 Comparison of the countermeasures for the memory dump

Card reference	Reading an address	Writing an address	Countermeasures	Memory dumped
a-21a	x	x		8000-FFFF
a-21b	x		Card Terminated	8000-FFFF
a-22a	x		Bypassed	8000-FFFF
a-22b	x	x		8000-FFFF
b-21a	x	x		8000-BFFF
b-22a	x	x		8000-BFFF
b-22b	x	x		8000-FFFF
c-22a	x		Partially bypassed	Seven bytes
c-22b			Strong	
d-22	x	x		8000-BFFF
e-21			Strong	

Table 5 Array bounds check

Card reference	Type confusion	Result after exceeding array's length
a-22a	Yes	6F 00
b-21a	Yes	6F 00
b-22b	Yes	6F 00
c-22a	Yes	6F 08

Table 6 Abusing transaction mechanism

Card reference	Call to new	Call to Make TransientArray	Type confusion
a-22a	No	Yes	Yes
b-21a	Yes	Yes	Yes
c-22a	No	Yes	No
e-21	No	Yes	No

same behaviour, it should be easy to get and exploit type confusion.

The other confusion we used is an array of bytes and an object. If we put a byte as first object attribute, it is bind to the array length. It is then really easy to change the length of the array using the reference to the object (Table 6).

As observed by [7] several cards refuse a code that creates a new object in a transaction. But surprisingly if we use the method of `MakeTransientArray` of the API it becomes feasible for the cards under test.

6 Future works

During the dump of some cards we discovered that we have had access to a RAM area at the lowest addresses in memory. Firstly, we discovered the reference of the APDU's class instance by using the same method as EMAN: 0x01D2 (situated in RAM area). At this address, the following structure was found: 00 04 29 FF 6E 0E. It represents the instance of APDU class, so we can deduce the address of the class APDU which is 0x6E0E (situated in ROM area).

After this observation, we wanted to find the APDU buffer in the RAM memory which is probably near to the class APDU's instance reference. That's why we have searched a table of 261 bytes (105 in hexadecimal). We found it at the address 0x1DC. It was confirmed because a pattern of an APDU command was found at the beginning of the table: "80 31 00 00 02".

Secondly, we wanted to find the stack. We believed it was near to the APDU buffer. So, we analyzed the operations used when the dump was made and looked in RAM memory. After that, we deducted that the stack was just before the APDU buffer, near to the address 0x7B. In fact, near to this we found this short value 0x01D2 which matches to the instance's reference of the APDU class and 0x01DC which is the address of the APDU buffer.

So we know that the Java stack is implemented on the a-21a card (no experiment has been conducted yet on the other cards). This means that we have the call history available and that the VM doesn't erase the value on top of the stack. We have to check if this is still valid during a context switch. If this hypothesis is true it becomes possible to steal the value of the PIN code of the user. It is passed as a parameter on top of the stack before the call to the method verifyPin().

Finally we want to evaluate the implementation of the BCV, because this component is known to be highly complex and prone to implementation errors. We are developing a model of this security function and a library to manipulate CAP file. Then thanks to our OPAL library,[1] we will generate test suites to characterize the BCV and check faulty implementation.

[1] http://gforge.inria.fr/projects/opal/

More generally, we believe that using various logical attacks could lead to information disclosure even for very recent smart cards. Furthermore, our next step is to use hardware attacks combined with logical attacks.

7 Conclusion

We are currently investigating other countermeasures as well as a way to remove one of our hypotheses. For instance, we are looking at the attack described in [7] to circumvent the BCV protection. Another way is to evaluate the implementation of the BCV which is a component known to be highly complex and prone to implementation errors. We believe that using a framework of different logical attacks should reveal BCV implementations flows.

We have shown in this paper the preliminary results of the EMAN attack. Our contribution is to propose a complete and optimized method of Hyppönen idea by allowing a self mutable code in a card, which has never been done until now. As a result, we are able to search and replace any code fragment in the memory, even if this memory segment is protected by the Java Card security mechanism. We have broken the segregation property offered by the firewall. This attack is based on two hypotheses: (i) post issuance is allowed and we have the necessary credentials, (ii) there is no BCV in the card. Our EMAN attack has been successfully tested against several smart cards. We demonstrated that this exploit was successful on some cards, only one became mute while detecting the ill-formed CAP file using a bytecode verifier. This attack has been made possible by the pre link process and the absence of an embedded bytecode verifier. In order to relax the hypothesis on the presence of such a bytecode verifier, we are investigating for another solution based on a hardware fault injection to introduce type confusion in the code stored in the EEPROM memory.

References

1. Virtual machine specification, java card platform, version 3.0, classic edition (2008). http://java.sun.com/javacard/3.0/
2. Global Platform Specification 2.2. http://www.globalplatform.org/specifications.asp
3. Girard, P., Lanet, J.L.: New security issues raised by open cards. Inf. Secur. Tech. Rep. **4**(1), 4–5 (1999)
4. Anderson, R., Kuhn, M.: Tamper resistance: a cautionary note. In: WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIXWorkshop on Electronic Commerce, p. 1. USENIX Association, Berkeley (1996)
5. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. IEEE **94**(2), 370–382 (2006)
6. Joint interpretation library application of attack potential to smartcards, v2.1, available at http://www.ssi.gouv.fr/site_documents/JIL/JIL-The_application_of_attack_potential_to_smartcards_V2-1.pdf (2006)
7. Mostowski,W., Poll, E.: Malicious code on java card smartcards: Attacks and countermeasures. In: Proceedings of the Smart Card Research and advanced application conference (CARDIS 2008), pp. 1–16 (2008)
8. Vertanen, O.: Java Type Confusion and Fault Attacks, Lecture Notes in Computer Science, vol. 4326/2006. pp. 237–251. Springer, Berlin (2006)
9. Witteman, M.: Smartcard security. Inf. Secur. Bull. **8**, 291–298 (2003)
10. Hyppönen, K.: Use of cryptographic codes for bytecode verification in smart card environment. Master's thesis, University of Kuopio (2003). Available at http://dx.doi.org/10.1007/978-3-540-69485-4_15